Александр Круглов

Ruby

Ruby 2.0.0p247

От автора

Текст книги в формате PDF доступен по адресу:
http://dl.dropbox.com/u/75172405/Ruby.pdf

Сборка:

1. Установить Ruby.
2. Установить необходимые пакеты - `gem install redcarpet`.
3. Скачать скрипт для преобразования Markdown в LaTeX
   http://gist.github.com/Krugloff/5491182.
4. Установить XeLaTeX.
5. Скачать стиль - http://gist.github.com/Krugloff/5491168.
6. Собрать pdf - `xelatex book.tex` (по умолчанию используются шрифты семейства Liberation).

Контакты:

- Исходники: http://github.com/Krugloff
- Автор: mr.krugloff@gmail.com
- Благодарности:
  - Qiwi: 89212870782
  - WebMoney: R349517838989

Оглавление

От автора **i**

Часть I. Основы 1

Глава 1. Основные понятия **2**
1.1. Краткое описание языка 2
1.2. Интерпретаторы 3
1.3. Краткое описание кода 4
1.3.1. Синтаксис выражений 5
1.3.2. Комментарии 6
1.4. Кодировка символов 7
1.5. Краткое описание ООП 9

Глава 2. Встроенные типы данных **13**
2.1. Простые типы данных 13
2.1.1. Числа (Numeric) 13
2.1.2. Текст (String) 14
2.1.3. Логические величины 16
2.1.4. Идентификаторы (Symbol) 16
2.1.5. Регулярные выражения (Regexp) 17
2.2. Составные типы данных 17
2.2.1. Индексные массивы (Array) 17
2.2.2. Ассоциативные массивы (Hash) 18
2.2.3. Диапазоны (Range) 19

Глава 3. Переменные и константы **20**

Глава 4. Выражения **23**
4.1. Операторы 23
4.2. Предложения 30
4.2.1. Условное предложение 30
4.2.2. Разветвленное условие 31
4.2.3. Цикл 32
4.2.4. Перебор элементов 33
4.2.5. Управление ходом выполнения 33
4.3. Триггеры 33

Глава 5. Реализация ООП **35**
5.1. Основные сущности 35
5.1.1. Модули (Module) 35
5.1.2. Классы (Class) 36
5.1.3. Константы 38
5.1.4. Переменные 38
5.1.5. Методы 39
5.1.6. Примеры 46
5.2. Основные принципы 48
5.2.1. Инкапсуляция 48
5.2.2. Наследование и агрегация 50
5.2.3. Полиморфизм 53

Часть II. Описание классов **55**

Глава 6. Числа **56**
6.1. Numeric 56
6.1.1. Приведение типов 56
6.1.2. Операторы 57
6.1.3. Округление 58
6.1.4. Математические функции 58
6.1.5. Предикаты 60
6.1.6. Итераторы 60
6.1.7. Остальное 60
6.2. Целые числа (Integer) 60
6.2.1. Приведение типов 61
6.2.2. Операторы: Fixnum и Bignum 61
6.2.3. Арифметические операции 62
6.2.4. Предикаты 62
6.2.5. Итераторы 63
6.2.6. Остальное 63
6.3. Десятичные дроби (Float) 63
6.3.1. Приведение типов 64
6.3.2. Операторы 64
6.3.3. Предикаты 65
6.3.4. Остальное 65
6.4. Рациональные дроби (Rational) 65
6.4.1. Приведение типов 65
6.4.2. Операторы 66
6.4.3. Остальное 66
6.5. Комплексные числа (Complex) 66
6.5.1. Приведение типов 67
6.5.2. Операторы 68
6.5.3. Математические функции 68
6.6. Math 69

Глава 7. Текст **72**
7.1. Текст (String) . . . 72
7.1.1. Приведение типов . . . 72
7.1.2. Элементы . . . 75
7.1.3. Операторы . . . 79
7.1.4. Изменение текста . . . 79
7.1.5. Поиск совпадений . . . 83
7.1.6. Предикаты . . . 86
7.1.7. Итераторы . . . 86
7.1.8. Кодировка символов . . . 87
7.1.9. Остальное . . . 87
7.2. Регулярные выражения . . . 88
7.2.1. Regexp . . . 88
7.2.2. MatchData . . . 90
7.3. Кодировка . . . 92
7.3.1. Кодировка текста (Encoding) . . . 92
7.3.2. Преобразование кодировок (Encoding::Converter) . . . 93

Глава 8. Составные объекты **98**
8.1. Array (индексные массивы) . . . 98
8.1.1. Приведение типов . . . 98
8.1.2. Элементы . . . 99
8.1.3. Операторы . . . 102
8.1.4. Изменение массивов . . . 103
8.1.5. Сортировка массива . . . 106
8.1.6. Итераторы . . . 107
8.1.7. Ассоциативные массивы . . . 108
8.1.8. Остальное . . . 108
8.2. Hash (ассоциативные массивы) . . . 110
8.2.1. Приведение типов . . . 110
8.2.2. Элементы . . . 111
8.2.3. Изменение массива . . . 112
8.2.4. Предикаты . . . 113
8.2.5. Итераторы . . . 114
8.2.6. Индексные массивы . . . 114
8.2.7. Остальное . . . 114
8.3. Range (диапазоны) . . . 115
8.3.1. Приведение типов . . . 116
8.3.2. Элементы . . . 116
8.3.3. Операторы . . . 116
8.3.4. Итераторы . . . 116
8.3.5. Остальное . . . 117
8.4. Enumerator (перечни) . . . 118
8.4.1. Приведение типов . . . 119
8.4.2. Элементы перечня . . . 119
8.4.3. Итераторы . . . 120

8.4.4. Остальное 120
8.4.5. Отложенные вычисления (Enumerator::Lazy) 120
8.5. Enumerable 122
8.5.1. Приведение типов 123
8.5.2. Элементы 123
8.5.3. Сортировка и группировка 123
8.5.4. Поиск элементов 124
8.5.5. Сравнение элементов 125
8.5.6. Предикаты 126
8.5.7. Итераторы 126
8.5.8. Циклы 127
8.5.9. Остальное 128

Глава 9. Объекты **129**
9.1. Интроспекция 129
9.1.1. Проверка выражений 129
9.1.2. Тип объекта 130
9.1.3. Иерархия наследования 131
9.1.4. Состояние объекта 132
9.1.5. Поведение объекта 133
9.2. Метапрограммирование 135
9.2.1. Выполнение произвольного кода 135
9.2.2. Вызов метода 136
9.2.3. Перехват выполнения 136
9.2.4. Изменение состояния 138
9.2.5. Изменение поведения 138
9.3. Остальное 140
9.3.1. Приведение типов 140
9.3.2. Сравнение объектов 141
9.3.3. Копирование объектов 141

Глава 10. Подпрограммы **145**
10.1. Замыкания (Proc) 145
10.1.1. Процедуры 146
10.1.2. Лямбда-функции 146
10.1.3. Использование замыканий 147
10.2. Методы 149
10.2.1. Method 149
10.2.2. UnboundMethod 151
10.3. Сопрограммы (Fiber) 152

Глава 11. Псевдослучайные числа (Random) **154**
11.1. Генераторы 155

Глава 12. Дата и время **156**
12.1. Время (Time) 156
12.1.1. Создание объекта 156
12.1.2. Приведение типов 157
12.1.3. Операторы 158
12.1.4. Форматирование 158
12.1.5. Системы остчета 158
12.1.6. Статистика 159
12.1.7. Предикаты 160
12.1.8. Остальное 161

Глава 13. Типы данных **162**
13.1. Логические величины 162
13.2. Symbol 162
13.2.1. Приведение типов 162
13.2.2. Операторы 163
13.2.3. Изменение регистра 163
13.2.4. Остальное 163
13.3. Структуры 164
13.3.1. Приведение типов 164
13.3.2. Элементы 165
13.3.3. Итераторы 165
13.3.4. Остальное 165

Часть III. Работа программы **167**

Глава 14. Выполнение программы **168**
14.1. Запуск программы 168
14.2. Ход выполнения 168
14.3. Завершение выполнения 169
14.4. Вызов системных команд 169

Глава 15. Чтение и запись данных **171**
15.1. IO (потоки) 171
15.1.1. Управление потоками 171
15.1.2. Приведение типов 174
15.1.3. Чтение данных 175
15.1.4. Запись данных 178
15.1.5. Стандартные потоки 180
15.1.6. Конвейеры 181
15.1.7. Мультиплексирование 182
15.2. File (файлы) 183
15.2.1. Взаимодействие с файловой системой 184
15.2.2. Путь к файлу 185
15.2.3. Права доступа 188

15.2.4. Статистика 189
15.2.5. Предикаты 190
15.3. Dir (каталоги) 191
15.3.1. Работа с файловой системой 192
15.3.2. Содержимое каталога 192
15.3.3. Итераторы 193
15.3.4. Остальное 193
15.4. Информация о файле 193
15.4.1. Класс File::Stat 194
15.4.2. Модуль FileTest 197

Глава 16. Обработка аргументов **199**
16.1. Файлы 199
16.1.1. ARGV 199
16.1.2. ARGF 199

Глава 17. Библиотеки кода **203**
17.1. Использование 203
17.2. Усовершенствование (Ruby 2.0) 204

Глава 18. Исключения **209**
18.1. Иерархия исключений 209
18.2. Методы 210
18.2.1. Exception 210
18.2.2. LoadError [ruby 2.0] 211
18.2.3. SignalException 211
18.2.4. SystemExit 211
18.2.5. Encoding::InvalidByteSequenceError 212
18.2.6. Encoding::UndefinedConversionError 212
18.2.7. StopIteration 212
18.2.8. LocalJumpError 213
18.2.9. NameError 213
18.2.10. NoMethodError 213
18.2.11. SystemCallError 213
18.3. Возникновение и обработка исключений 213
18.3.1. Вызов исключения 213
18.3.2. Обработка исключений 214
18.3.3. Catch и Throw 215

Глава 19. Тестирование и отладка **216**
19.1. Тестирование 216
19.1.1. Основы 216
19.1.2. TDD 217
19.2. Отладка 217
19.2.1. Состояние программы 217
19.2.2. Стек выполнения 218

19.2.3. Трассировка 220

Глава 20. Конкуренция и параллелизм 224
20.1. Основы 224
20.1.1. Параллелизм 224
20.1.2. Конкуренция 225
20.1.3. Состояние гонки 225
20.1.4. Современный параллелизм 226
20.2. Потоки выполнения (Thread) 226
20.2.1. Thread 227
20.2.2. Группировка потоков (ThreadGroup) 234
20.2.3. Синхронизация потоков (Mutex) 235
20.3. Процессы (Process) 236
20.3.1. Linux 236
20.3.2. Системные команды 237
20.3.3. Ruby 237
20.4. Обработка сигналов (Signal) 238
20.5. Событийная модель 239
20.6. Сбор мусора 240
20.6.1. Ruby 2.0 240

Глава 21. Безопасность 242
21.1. Уровни безопасности 242
21.1.1. Уровень 0 242
21.1.2. Уровень 1 242
21.1.3. Уровень 2 243
21.1.4. Уровень 3 243
21.1.5. Уровень 4 243
21.2. Модификаторы 244

Приложения 245

Приложение A. Запуск программы 246

Приложение B. Синтаксис регулярных выражений 250

Приложение C. Форматные строки 256

Приложение D. Присваивание 259

Приложение E. Преобразование кодировок 262

Приложение F. Упаковка данных 263

Приложение G. Форматирование времени 265

Приложение H. Создание потоков **270**

Приложение I. Файловая система Linux **272**
I.1. Типы файлов 272
I.2. Поиск файлов 273
I.3. Доступ к файлам 274
I.4. Открытие файла 276

Заключение **277**

Часть I

Основы

Глава 1

Основные понятия

Язык программирования - это искусственно созданный язык, облегчающий управление компьютером. Текст, написанный с помощью языка программирования называется кодом.

1.1. Краткое описание языка

> *Во имя эффективности - причем достигается она далеко не всегда - совершается больше компьютерных грехов, чем по любой другой причине, включая банальную глупость.*
>
> *W. A. Wulf*

Эта история началась 24 февраля 1993 года. Японский инженер Юкихиро Мацумото (Matz) решил создать новый язык программирования. Так появился Ruby.

> *Я хотел получить скриптовый язык, который был бы мощнее чем Perl, и поддерживал объектную парадигму лучше чем Python. Вот почему я решил разработать свой язык.*

Основное назначение Ruby — создание простых и в то же время понятных программ, где важна не скорость работы программы, а малое время разработки, понятность и простота синтаксиса.

Идеология Ruby:

- люди отличаются друг от друга, поэтому существует несколько способов решения одной задачи. Каждый выбирает тот способ, который ему по душе;
- опытные программисты не должны сталкиваться с неожиданным поведением своих программ (принцип наименьшего удивления);
- программирование должно приносить удовольствие. Все рутинные действия должны выполняться компьютером.

Когда распространение языка только начиналось, очень часто его упрекали за низкую скорость. В последних версиях скорость выполнения была значительно увеличена. Обычно говорят, что Ruby достаточно быстр - скорость выполнения компенсируется скоростью разработки. Медленный код при этом принято переписывать на Си. Си API - одна из полезных и удобных особенностей языка.

Особенности Ruby:

- интерпретируемый;
- объектно-ориентированный;
- строгая динамическая неявная типизация;
- любая синтаксическая конструкция относится к выражениям;
- гибкий и мощный синтаксис позволяет создавать программы, использующие термины предметной области (DSL);
- встроенная поддержка интроспекции и метапрограммирования;
- автоматический сбор мусора;
- возможность переопределения операторов;
- все классы доступны для изменения;
- встроенная поддержка Unicode;
- встроенная поддержка итераторов, подпрограмм и сопрограмм;
- возможность создания консольных, графических, мобильных и веб приложений.

1.2. Интерпретаторы

Интерпретатор - это программа, переводящая код в машинные команды, понятные компьютеру.

Интерпретаторы:

MRI - официальный интерпретатор, написанный на языке программирования Си и использующий виртуальную машину YARV (преимущество виртуальных машин в том, что код сначала интерпретируется полностью, и только затем выполняется).

Rubinius - сторонняя реализация виртуальной машины. Написана как с помощью языка программирования C++, так и самого Ruby;

JRuby - реализация языка для взаимодействия с виртуальной машиной Java;

IronRuby - реализация языка для взаимодействия с платформой .Net.

Официальный сайт языка. На нем можно узнать последние новости разработки интерпретатора, скачать исходный код и перейти к документации.

На этом сайте пользователи ОС Windows могут скачать установочный файл для своей операционной системы.

В состав официального интерпретатора входят:

- стандартная библиотека (наиболее часто используемые модули);

- менеджер пакетов RubyGems;
- интерактивный терминал irb (выполняет код на Ruby в режиме реального времени);
- генератор документации RDoc;
- программа ri для просмотра документации;
- менеджер задач Rake;
- шаблонизатор ERb.

1.3. Краткое описание кода

Программа на Ruby - это код, хранящийся в текстовом файле с определенным расширением (расширение файла - это группа символов, следующая за именем после точки).

Для языка Ruby используются два расширения: `.rb` (стандартное расширение) и `.rbw` (используется в Windows для создания программ с графическим интерфейсом).

Любой интерпретатор понимает только тот код, который написан по заранее определенным правилам. Правила бывают лексическими и синтаксическими.

Лексические правила регулируют определение существующих лексем в коде. Лексема - это последовательность символов, имеющая смысл для интерпретатора. Встретив в тексте набор символов, не относящийся к известным лексемам, интерпретатор завершит обработку кода и вернет сообщение об ошибке.

Типы лексем:

- элементарные типы данных - простейшие данные (числа, буквы, логические величины);
- идентификаторы - лексемы, использующиеся в двух случаях: для хранения результатов выполнения выражения и для пометки различных синтаксических структур. И то и другое необходимо для повторного применения кода или данных. Лексема идентификатора - это группа символов состоящая из букв, цифр и знаков подчеркивания (_). При этом идентификатор не может начинаться с цифры.

  Идентификаторы чувствительны к регистру. Интерпретатор по разному распознает строчные и прописные ASCII символы (принято использовать для идентификаторов именно ASCII символы, хотя в некоторых случаях это необязательное требование). Два идентификатора считаются идентичными только в том случае, если они состоят из одинакового набора байт;
- операторы - это знаки препинания, использующиеся в качестве разделителей и математические символы, позволяющие выполнять различные вычисления;

- инструкции - слова, зарезервированные языком программирования. Их переопределение невозможно, а использование приведет к заранее определенному результату. Инструкции используется для создания различных синтаксических структур или для управления процессом выполнения программы.

  Список инструкций: `__LINE__`; `__ENCODING__`; `__FILE__`; `__END__`; BEGIN; END; =begin; =end; alias; and; begin; break; case; class; def; defined?; do; else; elsif; end; ensure; false; for; if; in; module; next; nil; not; or; redo; rescue; retry; return; self; super; then; true; undef; unless; until; when; while; yield;

- комментарии - текст на естественном языке, поясняющий предназначение программы.

Синтаксические правила регулируют использование выражений в коде. Выражение - это синтаксическая единица, возвращающая результат после выполнения. Выражения могут быть простыми (состоящими из одной лексемы), сложными (состоящими из нескольких простых выражений, операторов или инструкций). Нарушение синтаксических правил приводит к завершению обработки кода интерпретатором.

Кроме жестких правил существуют также соглашения, принятые в сообществе. Они необязательны, но крайне желательны к выполнению. Следования правилам позволяет понимать ваш код компьютеру, а следование соглашениям облегчает его понимание для людей.

1.3.1. Синтаксис выражений

Для упрощения разработки в Ruby присутствует множество дополнений (дополнительные выражения) к синтаксису языка, которые делают его использование более удобным, но не добавляют новых возможностей. Такие дополнения обычно называют синтаксическим сахаром.

Минимальный набор синтаксических правил и соглашений описывает общие особенности употребления выражений в коде.

Соглашения:

- Код принято разбивать на строки. Каждая строка обычно не превышает 80 символов (это облегчает чтение кода);
- На каждой строке обычно располагается одно логически завершенное выражение (простое или сложное);
- Выражения обычно составляют таким образом, чтобы результат их выполнения мог быть использован несколько раз.
- Уровни вложенности выражений принято оформлять двумя пробелами.

Синтаксические правила:

- Для разделения отдельных выражений используется символ перевода строки (невидимый символ, добавляемый текстовым редактором автоматически, при нажатии клавиши ENTER);
- Для разделения выражений на одной строке используется точка с запятой (;);
- Если выражение не помещается на одной строке, то каждую строку, содержащую фрагмент выражения, заканчивают символом обратной косой черты (backslash - \), отделяя его пробелом (таким образом он находится прямо перед символом перевода строки и экранирует его). Всякий раз, когда необходимо визуально разбить выражение на несколько строк, а вы не уверены в правильности его обработки - используйте обратную косую черту;
- Строки, начинающиеся с точки (.), также считаются продолжением предыдущего выражения;
- Пробельные символы (пробел, отступ, перевод строки), не разделяющие выражения, игнорируются интерпретатором и могут использоваться для оформления кода. Однако стоит сохранять осторожность - в некоторых случаях они влияют на процесс выполнения программы.

1.3.2. Комментарии

Комментарии - это фрагменты текста на естественном языке, поясняющие цель написания кода. Комментарии должны использовать более высокий уровень абстракции (ближайший к человеческому мышлению) чем код.

Создание коротких и качественных комментариев - это одна из полезных способностей для программиста. Такие комментарии помогут разобраться в написанном коде не только другому разработчику, но и самому автору.

Обычно комментарий считается хорошим, если при перепроектировании кода комментарии к нему изменять не требуется.

Хоть наличие комментариев и облегчает понимание кода, главный вклад все же вносится хорошим стилем программирования и следование соглашениям. Никакой комментарий не спасет плохо написанной программы.

Следует помнить, что кроме пользы комментарии также приносят и вред - нарушают визуальное оформление кода.

Комментарии не обрабатываются интерпретатором и не влияют на процесс выполнения программы.

В Ruby существует два способа создания комментариев:

- Любой текст, начинающийся символом решетки и заканчивающийся переводом строки, считается комментарием.

```
# Это комментарий.
# Это тоже комментарий.
```

- Любой текст между инструкциями `=begin` и `=end` на отдельных строках считается комментарием. Текст комментария начинается после первого пробельного символа.

```
=begin Это тоже комментарий.
В нем можно записывать все что угодно.
Обычно его называют встроенной документацией,
а на первой строке записывают название программы
для ее обработки.
=end
```

1.4. Кодировка символов

Текст, который сохраняется в файле, для компьютера существует не в виде символов, а только как двоичные данные - числа из нулей и единиц. Каждый ноль или единица занимают один бит памяти. Восемь нулей или единиц занимают один байт. Обычно удобно записывать байты используя шестнадцатеричную систему счисления. Так ff в шестнадцатеричной системе соответствует 255 в десятичной и 11111111 в двоичной.

Каждому символу соответствует определенное число, хранящееся в памяти компьютера. Это число также называется кодовой позицией (code point) символа. Таблица, в которой соотносятся кодовые позиции и символы называется кодовой таблицей.

Кодировка - это способ представления символов в памяти компьютера (в виде набора байт). Например пробел хранится как 10000 (32 в десятичной системе счисления, 20 в шестнадцатеричной). Без информации об используемой кодировке компьютер не сможет правильно отобразить сохраненный набор байт. Обычно понятия кодировки и кодовой таблицы взаимозаменяемы.

В начале появления компьютеров повсевместно использовалась кодировка ASCII, включающая кодовые позиции для 127 символов: цифр, знаков, букв латинского алфавита и спецсимволов. Для предоставления 127 различных кодовых позиций хватает 7 бит памяти, поэтому кодовая позиция в ASCII полностью аналогична байту, хранящемуся в памяти.

В качестве минимальной единицы памяти обычно используется один байт. В одном байте может быть сохранено 255 различных кодовых позиций. Оставшиеся 128 чисел (от 128 до 255) использовались для представления национальных символов: букв национальных алфавитов и специфичных знаков.

Например слово hello хранится в памяти в виде набора байт 48 65 6c 6c 6a. Каждый байт одновременно является кодовой позицией символа.

Обилие национальностей и ограниченный набор различных вариаций битов привели к образованию огромного количества кодировок. Каждая кодировка по разному использовала оставшиеся кодовые позиции, представляя с их помощью разные символы. Это создавало сразу две проблемы: преобразования кодировок и ограниченности набора символов.

Для решения проблемы кодировок был создан стандарт Юникод (Unicode). Юникод был попыткой создать единый набор символов, который будет содержать в себе все символы всех языков на планете. В стандарте определены кодовые позиции символов, но не способ их хранения. Правила, согласно которым кодовые позиции преобразуются в байты (машинное представление), определяются Юникод-кодировками.

Кодовая позиция символа в Юникод записывается в формате U+xxxx, где x - это цифры в шестнадцатеричной системе счисления (может использоваться больше четырех цифр). Количество возможных кодовых позиций превышает миллион, что позволяет стандартизировать большинство существующих алфавитов. На данный момент стандарт содержит кодовые позиции около 100 тыс. символов.

Например, слово hello состоит из пяти кодовых позиций: U+0048 U+0065 U+006C U+006C U+006F.

Разные Юникод-кодировки довольно сильно отличаются. Слово hello может быть закодировано как в виде набора байт 00 48 00 65 00 6C 00 6C 00 6A, так и виде 48 00 65 00 6C 00 6C 00 6A 00. Существуют также кодировки, хранящие каждую кодовую позицию в четырех байтах.

В последнее время чаще всего используется Юникод-кодировка UTF-8. Она совместима с ASCII - для кодирования каждого символа, содержащегося в ASCII используется один байт (слово hello в UTF-8 кодируется так же как и в ASCII). Остальные символы кодируются двумя и более байтами. Это позволяет не хранить в памяти байты, содержащие только нули, и правильно обрабатывать ASCII символы.

Приступая к выполнению программы, интерпретатор получает лишь набор байт. В зависимости от внутренней кодировки эти байты могут интерпретироваться по разному. По умолчанию интерпретатор считает, что внутри программы используется кодировка ASCII (**во второй версии Ruby** - UTF-8, поэтому в ней для изменения кодировки чаще всего нет необходимости). Все лексические правила также определены относительно символов, содержащихся в ASCII.

Если в коде или комментариях (ведь это тоже одна из лексем) используются символы, не входящие в ASCII (например кириллица), необходимо вручную указывать кодировку программы.

Если текстовый редактор сохраняет код в кодировке, отличной от ASCII, то она также должна быть явно указана в качестве кодировки программы.

Кодировка устанавливается с помощью специального комментария, расположенного в самом начале программы: `#coding: название_кодировки`

Существует несколько отдельных лексических правил для такого комментария:

- Вместо coding также может быть использовано encoding;
- Вместо двоеточия также может использоваться знак равенства;
- Пробелы до и после двоеточия игнорируются;
- Весь комментарий не чувствителен к регистру;
- Перед coding также может использоваться набор символов -*-.

1.5. Краткое описание ООП

Парадигма программирования - это система идей и понятий, определяющих стиль написания компьютерных программ.

Парадигмы программирования не являются взаимоисключающими и, следовательно, могут сочетаться.

Объектно-ориентированная парадигма (ООП) - это парадигма программирования, в которой основными концепциями являются понятия объектов и классов. Большинство основных положений ООП было развито в языке программирования Smalltalk, сильно повлиявшем на Ruby. В настоящее время количество прикладных языков программирования, реализующих ООП, преобладает.

Хоть ООП и является достаточно удобной парадигмой, не стоит забывать, что она далеко не единственная. В некоторых случаях применение объектного подхода не оправдано и снижает производительность и удобство создания программ. В любом случае необдуманное и чрезмерное применение концепций ООП усложняет и замедляет выполнение кода, превращая его в бесполезный, малопонятный набор выражений.

ООП развивает идеологию процедурного программирования, где данные и подпрограммы (процедуры или функции) их обработки формально не связаны.

ООП - это не только набор конкретных методик, а также еще и философия проектирования приложений. Как и любая сложная парадигма, ООП состоит из нескольких уровней понимания. Следует заметить, что применение объектно-ориентированного языка Ruby не означает, что код автоматически становится объектно-ориентированным - требуется явная реализация и использование объектно-ориентированных концепций.

Основные понятия ООП - это абстракция, класс, объект, свойство, метод, инкапсуляция, наследование и полиморфизм.

Объектно-ориентированный дизайн - это дисциплина, описывающая способы (варианты) определения объектов и их взаимодействия для решения проблемы, которая определена и описана в ходе объектно-ориентированного анализа. Поведение программы формируется с помощью группы объектов, обменивающихся сообщениями для взаимодействия.

Абстракция:

Абстрагирование - это способ выделить существенные свойства и игнорировать несущественные. Соответственно, абстракция - это набор выделенных существенных свойств.

Существенные свойства - это свойства, которыми сущность обязана обладать, чтобы быть именно этой сущностью. Несущественные свойства - свойства, обладание которыми необязательно.

С точки зрения сложности, главное достоинство абстракции в том, что она позволяет игнорировать несущественные детали (не имеющие значения для программы). Абстракция - это один из главных способов борьбы со сложностью реального мира.

Класс:

Класс - это абстрактная (виртуальная) модель (абстрактный тип данных), еще несуществующей сущности. Фактически класс является образцом для создания новых объектов или сущностей (формулой или руководством по эксплуатации). Обычно классы относятся к статичным сущностям, существующим в коде и неизменным в процессе выполнения.

В отличии от объектов, классы обычно не содержат данных. Передача данных классу позволяет создавать объекты, описанного в классе типа (в ООП понятия тип данных и класс - синонимы).

Объект, созданный по образцу класса называют экземпляром этого класса. Основное предназначение классов - определять поведение своих экземпляров. Обычно классы создают таким образом, чтобы они описывали объекты предметной области (объекты реального мира).

Классы позволяют описывать одинаковые сущности только один раз, уменьшая этим размеры и сложность программы.

Объект:

В основе ООП находится понятие объекта. Объект - это абстракция изменяемого состояния памяти компьютера. Обычно объекты относятся к динамичным сущностям, создаваемым и изменяемым в процессе выполнения программы.

Объекты обладают состоянием и поведением.

Состояние объекта зависит от значения его свойств (хранимых объектом данных).

Поведение объекта зависит от набора доступных ему методов. Методы - это сообщения, которыми обмениваются объекты. Это абстрактные сущности, определяющие действия, которые можно выполнить над объектом и действия, которые сам объект может выполнять. Пользуясь методами объекта мы можем влиять на его состояние, а посылая сообщения классу мы можем изменять состояние всех его экземпляров.

Таким образом ООП оперирует состоянием, заключенным внутри объекта, и позволяет влиять на него с помощью предоставленных методов.

Создание объектов позволяет уменьшить сложность программы, акцентируя внимание только на использующихся сущностях и их взаимодействии.

Инкапсуляция:

Инкапсуляция - это механизм языка, позволяющий сущности объединять в себе данные и методы для работы с этими данными. Данные при этом скрыты от остальной программы, а методы доступны для взаимодействия объектов. Объект не считается отдельной сущностью, если его состояние может быть изменено без явного использования ссылки на объект.

- Инкапсуляция позволяет распараллелить процессы создания программы, ускоряя разработку ПО;

- Инкапсуляция снижает сложность разработки, позволяя сосредоточиться на небольших фрагментах программы;
- Инкапсуляция помогает сокрытию деталей реализации, необходимых программе, но выходящих за рамки абстракции. Инкапсуляция помогает управлять сложностью, скрывая доступ к ней.

Наследование:

Наследование - это механизм языка, позволяющий сущности использовать структуру другой сущности, заимствуя и расширяя уже имеющуюся функциональность (например классы расширяют возможности модулей). Класс, который заимствуется (наследуется) называется базовым или суперклассом, а класс, который заимствует (наследует) - производным или подклассом. Все базовые и производные классы в общем создают иерархию классов программы.

- Наследование снижает время на разработку за счет повторного использования кода;
- Наследование снижает сложность, позволяя использовать уже известные фрагменты программы. Однако при сложной иерархии повышается объем кода, с которым работает программист в отдельный момент времени;
- Наследование дополняет абстракцию, выделяя сущности с незначительным уровнем различий. Наследование позволяет создавать абстракции с различным уровнем реализации (дополнительными группами существенных свойств).

Полиморфизм:

Полиморфизм - это механизм языка, позволяющий производным классов изменять унаследованное поведение с сохранением общей структуры.

- Полиморфизм повышает скорость разработки, позволяя быстро подстраиваться под требования заказчика;
- Полиморфизм снижает сложность, позволяя скрывать внутреннюю структуру объектов;
- Полиморфизм поддерживает возможность отдельной реализации базовых методов для производных классов.

Типизация данных:

> *”Если что-то выглядит как утка, плавает как утка и крякает как утка, то, вероятно, это утка”.*

В Ruby применяется строгая динамическая неявная типизация.

При строгой типизации совместимость и границы использования типа объекта контролируются интерпретатором и каждый объект имеет тип (использование

объектов не подходящего типа считается исключением и приводит к завершению выполнения программы).

При динамической (полиморфной) типизации тип объекта вычисляется во время выполнения и может произвольно изменяться в процессе. Динамическая типизация облегчает реализацию полиморфизма.

При неявной (утиной) типизации (подвид динамической типизации) совместимость и границы использования объекта ограничены его текущим набором методов и свойств, в противоположность наследованию от определенного класса. Утиная типизация ставит во главу угла не тип объекта, а его возможности.

Для того чтобы в Ruby узнать возможность использования того или иного объекта проверяется не его класс, а его реакция на вызов определенных методов в текущий момент. Если реакция объекта удовлетворяет условию, то его использование разрешается. При прочтении этой книги вы встретите множество условий, требующих определенной реакции объекта, на вызов того или иного метода.

Глава 2

Встроенные типы данных

Каждый язык программирования поддерживает один или несколько встроенных типов данных. Экземпляры стандартных типов создаются без явного указания класса создаваемого объекта - он вычисляется автоматически на основе существующих лексических правил.

2.1. Простые типы данных

К простым типам данных относятся числа, текст, логические величины, идентификаторы и регулярные выражения. Такие типы данных - это базовые блоки для построения других типов. На их основе создаются все остальные классы.

2.1.1. Числа (Numeric)

Целые числа (Integer)

- Целые числа, занимающие в памяти не более 31-го бита, относятся к классу Fixnum.
- Целые числа, превышающие этот размер, относятся к классу Bignum.
- Преобразование между типами чисел происходит автоматически.

Лексема числа - это обычный набор цифр (1289). Для разделения разрядов может использоваться символ подчеркивания (_), который будет игнорироваться интерпретатором. Однако этот знак нельзя использовать в начале или в конце лексемы (`1_000_000` - соответствует одному миллиону).

Системы счисления:

- По умолчанию, все числа обрабатываются в десятичной системе счисления. Результат любых вычислений также преобразуется в десятичную систему;
- Числа, начинающиеся с приставки 0x или 0X, обрабатываются в шестнадцатеричной системе счисления (0x4AF);
- Числа, начинающиеся с приставки 0b или 0B, обрабатываются в двоичной системе счисления (0b0111).

Для записи отрицательных чисел используется знак ”минус” (-).

Для записи положительных чисел используется знак ”плюс” (+). По умолчанию все числа обрабатываются как положительные.

Десятичные дроби (Float)

Лексема десятичной дроби - это группа цифр, разделенных десятичной точкой на две части: целую и дробную (`123.051`).

Так же можно использовать научную или экспоненциальную нотацию. При этом после числа записывается символ экспоненты (e или E), после которого следует отрицательное или положительное число, обозначающее показатель степени 10 (`123e-10` - соответствует $123 * 10^{-10}$).

При записи десятичных дробей для разделения разрядов может использоваться символ подчеркивания (_), который будет игнорироваться интерпретатором.

2.1.2. Текст (String)

Текст - это набор из одного или более символов. Символ - это любой отображаемый на экране знак.

Обычно этот тип данных называют строками. Несмотря на то, что текст может содержать символ перевода строки, он все равно рассматривается как одна большая строка. Я решил использовать термин "текст" чтобы избежать путаницы между понятиями строки кода (line), строки как объекта (string) и строки текста. В английском языке для каждой из строк существует отдельный термин, который при переводе на русский теряет смысловую нагрузку.

Простой текст: группа символов, ограниченная одиночными кавычками (`'Ruby'`).

Простой текст обрабатывается в том виде, в котором записан.

В простом тексте также распознается минимальный набор спецсимволов (называемых также управляющими или экранированными последовательностями). Спецсимвол - это группа из одного или более символов, теряющих своё индивидуальное значение, одновременно с приобретением этой группой нового значения.

Спецсимволы:

\' - соответствует символу одиночной кавычки;
\ - соответствует символу обратной косой черты.

Составной текст: группа символов, ограниченная двойными кавычками (`"Ruby"`).

Составной текст обрабатывается интерпретатором с распознаванием полного набора спецсимволов и поддержкой интерполяции.

Интерполяция - это выполнение фрагментов кода `"#{выражение}"` и замена их на результат выполнения выражения (`"#{1+2}"` - соответствует тексту "3").

В составном тексте также разрешается использовать другие парные символы двойных кавычек.

Спецсимволы:

* - соответствует любому символу на месте *, который необходимо сохранить в тексте. Используется для экранирования символов. Поэтому спецсимволы также называют экранированными последовательностями - все они начинаются с обратной косой черты;

\b - удаление предыдущего символа;

\r - возврат указателя курсора на начало строки. Запись следующего символа удалит все предыдущие (символ возврата каретки). Спецсимвол используется для вставки новой строки вместо предыдущей. Это полезно для замены строк в различных программах (например отображение хода выполнения программы в терминале).

\n - перевод указателя курсора на начало новой строки (символ перевода строки). Для операционной системы Windows в качестве символа перевода строки используется спецсимвол `\r\n`.

\t - перевод указателя курсора вправо, создавая отступ (табуляцию);

*** - соответствует символу, с указанной кодовой позицией из трех цифр в восьмеричной системе счисления;

** - соответствует `\0**`;

* - соответствует `\00*`;

\x** - соответствует символу, с указанной кодовой позицией;

\x* - соответствует `\x0*`;

\u**** - соответствует символу, с указанной кодовой позицией в стандарте Юникод;

\u{*} – соответствует группе символов, с указанными кодовыми позициями в стандарте Юникод.

Специальная форма записи: приставки `%q` или `%Q`.

Текст также может быть записан между двумя произвольными разделителями с использованием приставок `%q` или `%Q`. Разделитель – это символ или группа символов, которая служит границами текста. При использовании приставки `%q` текст будет распознаваться как простой, а при использовании приставки `%Q` - как составной (`%Q(Ruby)` - соответствует тексту `Ruby`). Вместо приставки `%Q` также можно использовать только знак процента (`%(Ruby)`).

Документы: большие блоки текста, с многочисленными знаками препинания.

Лексема документа начинается с символов `<<` или `<<-`. За ними следует группа символов, которая будет служить границей текста.

Тело документа (текст) начинается со следующей строки. Объект создается, когда на отдельной строке будет использован разделитель. После создания объекта интерпретатор продолжит обработку кода с того места, на котором встретил начало лексемы.

- Когда лексема начинается с <<, пробелы между началом строки и конечным разделителем не допускаются. Пробелы после конечного разделителя не допускаются никогда;
- Когда начальный разделитель не ограничен кавычками, лексема распознается как составной текст. Чтобы сохранить простой текст начальный разделитель ограничивают одинарными кавычками. Кавычки также позволяют использовать пробелы внутри разделителя (в данном случае в качестве разделителя будет выступать объект).

```
<<- 'DOC'
  Здесь записан простой текст.
DOC
# -> 'Здесь записан простой текст'
```

Одиночный символ: символ, начинающийся со знака вопроса ?.

При использовании лексемы распознаются некоторые спецсимволы, в основном относящиеся к способам записи символов с помощью кодовых позиций (`?A` – соответствует тексту 'A').

2.1.3. Логические величины

Логические величины используются для булевой алгебры, проверки различных условий или сравнении объектов.

Список лексем:

Истина: true, ссылается на единственный экземпляр класса TrueClass;
Ложь: false, ссылается на единственный экземпляр класса FalseClass;
Отсутствие: nil, ссылается на единственный экземпляр класса NilClass. Используется в том случае, если необходимо представить отсутствие объекта, подходящего под заданные условия.

При использовании в выражениях лексемы false и nil имеют логическое значение false, а все остальные объекты - логическое значение true.

Иногда также говорят о положительном результате (логическое значение true) и отрицательном (логическое значение false).

2.1.4. Идентификаторы (Symbol)

Довольно часто для управления программой используются небольшие группы символов. Из-за особенностей реализации использование для этого текстовых объектов снижает скорость выполнения программы. Вместо этого рекомендуется использовать экземпляры класса Symbol.

Лексема объекта-идентификатора - это группа символов, следующая за двоеточием (`:green`). Также идентификатор может быть записан между двумя произвольными разделителями, с использованием приставки `%s`

(`%s(Ruby)` соответствует объекту-идентификатору `:Ruby`)

Для каждого идентификатора, кроме текста, сохраняется также цифровой код, вычисленный на его основе. При повторном использовании той же группы символов, вместо создания нового объекта, по цифровому коду будет найден уже существующий. Использование цифровых кодов ускоряет поиск и сравнение объектов.

Однажды созданный объект-идентификатор будет существовать до закрытия программы, поэтому необходимо осторожно подходить к их использованию и делать это только по назначению (с целью управления процессом выполнения программы). Динамическое создание множества идентификаторов увеличивает количество памяти, занимаемой программой.

2.1.5. Регулярные выражения (Regexp)

Регулярные выражения - это мощный инструмент для поиска по тексту. С помощью регулярных выражений составляются образцы, на основе которых выполняется поиск.

Лексема регулярного выражения - это группа символов (называемая телом регулярного выражения), ограниченная двумя косыми чертами (slash - /). После конечного разделителя может быть использован необязательный модификатор, влияющий на механизм поиска (`/Ruby/i`).

Тело регулярного выражения также может быть записано между двумя произвольными разделителями с использованием приставки `%r`. Модификаторы в этом случае записываются после конечного разделителя (`%r(Ruby)i` – соответствует `/Ruby/i`).

Тело регулярного выражения обрабатывается как составной текст. Одиночные символы, не относящиеся к спецсимволам соответствуют их аналогам в тексте. Поиск совпадений выполняется последовательно по каждому символу, слева направо.

Полный синтаксис регулярных выражений описывается в приложении.

2.2. Составные типы данных

Составные объекты - это объекты, содержащие произвольный набор элементов (любых других объектов). Составные типы позволяет группировать объекты и рассматривать их на уровне группы.

2.2.1. Индексные массивы (Array)

Индексный массив (или просто массив) - это составной объект, содержащий упорядоченную группу элементов и позволяющий получить доступ к элементу,

если известна его позиция (индекс элемента). Элементами массива могут быть любые объекты (даже другие массивы).

С точки зрения синтаксиса, индексные массивы - это группа объектов между двумя квадратными скобками. Сами объекты при этом разделяются запятыми.

```
[1, "Ruby", ?\u0048]
```

Существует также специальный синтаксис записи массивов, в качестве элементов которых выступают короткие отрывки текста (состоящие из одного слова и не содержащие пробелов). Такие массивы могут быть записаны в виде группы элементов между двумя произвольными разделителями с использованием приставок `%w` или `%W`. Сами элементы при этом разделяются пробелами.

При использовании приставки `%w` элементы массива будут рассматриваться как простой текст, а при использовании приставки `%W` - как составной.

`%W( Язык программирования Ruby )` соответствует массиву:

```
[ "Язык", "программирования", "Ruby" ]
```

Во второй версии Ruby добавлен специальный синтаксис записи массивов, в качестве элементов которых выступают объекты-идентификаторы. Такие массивы могут быть записаны в виде группы элементов между двумя произвольными разделителями с использованием приставок `%i` или `%I`. Сами элементы при этом разделяются пробелами.

При использовании приставки `%i` элементы массива будут рассматриваться как простой текст, а при использовании приставки `%I` - как составной.

`%I( category klass )` соответствует массиву:

```
[:category, :klass]
```

2.2.2. Ассоциативные массивы (Hash)

Ассоциативный массив - это составной объект, содержащий упорядоченную группу элементов (каждый из которых представляет собой пару ключ/значение) и позволяющий получить доступ к значению, если известен его ключ (ключ ассоциируется с объектом).

Для построения соответствий между ключами и значениями используется виртуальная таблица.

Объекты, выступающие в роли ключей, в таблице представлены в виде цифровых кодов (небольших целых чисел), получаемых в результате вызова метода ‘.hash‘. Соответственно сам ключ может быть объектом любого типа, если для него определен этот метод.

В ассоциативном массиве можно хранить любые объекты (даже другие массивы).

С точки зрения синтаксиса, ассоциативные массивы - это группа парных элементов ключ/значение между двумя фигурными скобками. Ключи от значений отделяются символами =>. Сами элементы при этом разделяются запятыми.

```
{ "Ruby" => "language", "Вася" => "Человек" }
```

Один из наиболее распространенных способов использования объектов-идентификаторов - в качестве ключей ассоциативного массива. При этом ассо-

циативный массив записывают в коде программы, отделяя ключи от значений двоеточием (при этом двоеточие перед ключом не используется).

```
{ Ruby: "language", Вася: "Человек" }
```

соответствует ассоциативному массиву:

```
{ :Ruby => "language",  :Вася => "Человек" }
```

2.2.3. Диапазоны (Range)

Диапазон - это составной тип данных, содержащий упорядоченный набор объектов, располагающихся между заданными границами.

С точки зрения синтаксиса, диапазоны - это два однотипных объекта, разделенные двумя или тремя точками. При использовании двух точек в диапазон включается конечная граница (`1..3` содержит числа 1, 2, 3), а при использовании трех - нет (`1...3` содержит числа 1 и 2).

Границы диапазона должны принадлежать к одному классу. В этом классе должен быть определен оператор '<=>', использующийся для сравнение объектов, входящих в диапазон, с его границами.

Глава 3

Переменные и константы

Переменные и константы - это идентификаторы, предназначенные для получения доступа к объектам. Объекты, имеющие идентификатор могут быть использованы повторно. Выражение, связывающее объекты и идентификаторы, называется выражением присваивания. Объект, на который ссылается идентификатор, считается его значением.

Переменные и константы также позволяют логически разделить используемые данные. Они облегчают понимание кода, позволяя перейти от терминов языка программирования к терминам решаемой задачи (проблемной области). Адекватность переменной (константы) во многом определяется ее названием. Название переменной можно рассматривать как высокоуровневый псевдокод, характеризующий ее содержимое.

Константы от переменных отличаются только областью применения. Константы используются для доступа к единственному постоянному объекту. В отличии от констант переменные подразумевают многократное использование. Переменная может использоваться как для изменения текущего объекта, так и для изменения значения переменной.

На самом деле, в Ruby, переопределение констант не приведет к завершению процесса выполнения программы. Вместо этого интерпретатор просто выведет обычное предупреждение.

Для использования переменных и констант необходимо объявить интерпретатору об их существовании. Переменные и константы считаются существующими после выполнения первого выражения присваивания с их участием. Это выражение называют инициализацией. Процесс инициализации состоит из объявления (создания) идентификатора и определения (присваивания) объекта.

Идентификаторы также могут быть объявлены и без выражения присваивания. Встретив подходящую лексему интерпретатор создаст требуемую переменную или константу. В этом случае определение выполняется автоматически (значением становится nil). Автоматическая инициализация предотвращает ошибки, возникающие при использовании переменных, не имеющих значения. Она также позволяет акцентировать внимание на объектах, а не на переменных.

Идентификатор считается объявленным в любом случае если код содержит ее лексему, даже если фрагмент кода не выполнялся (значением становится nil). Это происходит из-за предварительной обработки кода для виртуальной машины.

Обычно если переменная объявлена, но при этом нигде не используется, то интерпретатор выдаст предупреждение. Чтобы этого избежать в качестве

неиспользуемой переменной объявляют '_'. Во второй версии Ruby любая переменная, начинающаяся с подчеркивания будет считаться не используемой и предупреждений не будет.

Область видимости:

Фрагмент кода, в котором идентификатор существует и может быть использован.

На основе областей видимости также осуществляется классификация переменных. К базовым областям видимости относятся глобальная и локальные области. Глобальная область видимости распространяется на весь код. Глобальные переменные и константы существуют в любом месте кода (после их инициализации). Локальная область видимости распространяется только на явно ограниченный фрагмент кода. Локальные переменные и константы существуют только в той части кода, в которой происходила их инициализация.

- Лексемы локальных переменных начинаются с подчеркивания или строчной буквы (принято использовать только строчные буквы, разделяя слова знаком подчеркивания - змеиная_нотация). Использование несуществующей локальной переменной считается вызовом метода;
- Лексемы глобальных переменных начинаются с знака доллара. Использование несуществующей глобальной переменной считается её объявлением;
- Лексемы констант начинаются с прописной буквы (принято использовать только прописные буквы, разделяя слова знаком подчеркивания - НОТАЦИЯ_ГРЕМУЧЕЙ_ЗМЕИ). Использование несуществующей константы считается исключением (поиск констант включает области видимости верхнего уровня).

ООП вводит две дополнительные области видимости: область видимости класса и область видимости экземпляра класса. Так же добавляется два вида переменных.

- Лексема переменной экземпляра начинается с знака @ (принято использовать змеиную нотацию). Использование несуществующей переменной экземпляра считается её объявлением;
- Лексема переменной класса начинается с @@ (принято использовать змеиную нотацию). Использование несуществующей переменной класса считается исключением.

Сбор мусора:

В Ruby ресурсы, используемые программой, управляются интерпретатором. Это позволяет избегать наиболее распространенных проблем работы с памятью. В то же время интерпретатор распознает только заранее определенные ситуации, что иногда приводит к довольно неприятным результатам. Дополнительная работа также увеличивает время интерпретации и выполнения программ. Баланс

между этими двумя полюсами - важная задача для разработчиков интерпретатора.

Для автоматического управления памятью реализован механизм, называемый сбором мусора. Под мусором подразумеваются объекты, сохраненные в памяти, но при этом не использующиеся. Как только интерпретатор понимает, что объект не связан ни с одним идентификатором, он освобождает память, которую этот объект занимал (потому что доступ к этому объекту больше невозможен). Это позволяет удалять одноразовые объекты сразу после их использования, а также сохранять синтаксические структуры на всем протяжении процесса выполнения.

Глава 4

Выражения

Выражение - это команда, возвращающая результат выполнения (обычно это объект) в место своего вызова. Выражения делятся на простые и сложные. Стандартные объекты и идентификаторы относятся к простым выражениям. Для выполнения вычислений из простых выражений создаются сложные выражения и предложения (которые являются одной из разновидностей сложных выражений). Части сложного выражения соединяются с помощью операторов и инструкций.

4.1. Операторы

Простейший способ создания сложных выражений - объединение простых с помощью операторов. Операторы - это группа математических символов или знаков препинания. Составные части сложного выражения называют операндами. Операнды, в свою очередь, также относятся к выражениям и могут быть как простыми выражениями, так и сложными.

Классификация:

- Унарные (У) - оперируют одним операндом;
- Бинарные (Б) - оперируют двумя операндами;
- Тернарные (Т) - оперируют тремя операндами.

Операторы также отличаются приоритетом и последовательностью вычисления операндов.

В сложных выражениях, содержащих несколько операторов, операнды будут вычисляться в порядке увеличения приоритета их операторов.

Если существует несколько операторов с одинаковым приоритетом (или только один оператор), то операнды вычисляются в том порядке, в котором были записаны. При R-последовательности это будет происходит справа налево, а при L-последовательности - слева направо.

Чтобы изменить процесс выполнения выражения, операнды, которые необходимо вычислить в первую очередь, отделяют двумя круглыми скобками.

Приор.	Лексема	Послед.	Тип	Название выражения
1	! ~ +	R	У Б У	логическое отрицание; побитовое отрицание; унарный плюс
2	**	R	Б	возведение в степень
3	-	R	У	унарный минус
4	* / %	L	Б	произведение (копирование); деление; остаток от деления (форматирование)
5	+ -	L	Б	сложение (объединение); вычитание (удаление)
6	<< >>	L	Б	побитовый сдвиг влево (добавление) побитовый сдвиг вправо
7	&	L	Б	побитовое И (пересечение множеств)
8	| ^	L	Б	побитовое ИЛИ (объединение множеств); побитовое исключающее ИЛИ
9	< <= > >=	L	Б	отношение
10	<=> ! = =~	L	Б	сравнение; неравенство; поиск совпадений
10	!~ == ===	L	Б	отсутствие совпадений; равенство; проверка условия
11	&&	L	Б	логическое И
12	||	L	Б	логическое ИЛИ
13	?:	R	Т	логическое условие
14	=	R	Б	присваивание
15	not	R	У	логическое отрицание
16	and or	L	Б	логическое И; логическое ИЛИ

1. **!obj** (логическое отрицание)

   Используется для получения противоположного логического значения.

   ```
   !1 # -> false
   !nil # -> true
   ```

   ~ integer (побитовое отрицание)

   Каждый бит числа изменяется на противоположный и дополняется до 1. В результате возвращается десятичное число, необходимое для дополнения. Аналогично выполнению выражения `-number-1`.

   ```
   ~1 # -> -2
   ~0b01 # -> -2
   ```

   + number (унарный плюс)

   Возвращается число в десятичной системе счисления.
   `+0b01 # -> 1`

2. **number**number** (возведение в степень)

   Используется для возведения числа в степень. Первое операнд - основание степени, а второй - показатель.
   `2**3 # -> 8`

3. **- number** (унарный минус)

   Используется для получения числа, противоположного по знаку.
   `-0b01 # -> -1`

4. **number * number** (произведение)

   Используется для перемножения двух чисел.
   `1 * 2 # -> 2`

   string * integer (копирование текста)

   Используется для копирования фрагмента текста заданное число раз.
   `"R" * 3 # -> "RRR"`

   array * integer (копирование массива)

   Используется для копирования фрагмента массива заданное число раз.
   `[1, ?R] * 2 # -> [1, "R", 1, "R"]`

   [*object] (извлечение элементов)

   Используется для извлечения элементов составного объекта.

```
a = [1, 2, 3]
[*a] # -> [1, 2, 3]
[*a, 1] # -> [1, 2, 3, 1]
b = { a: 1, b: 2 }
[*b] # -> [ [:a, 1], [:b, 2] ]
c = 1..4
[*c] # -> [1, 2, 3, 4]
[*1] # -> [1]
[*nil] # -> [ ]
[*?a] # -> ["a"]
```

number / number (деление)

Используется для деления двух чисел.

```
-6 / 3 # -> -2
```

number % number (остаток от деления)

Используется для целочисленного деления двух чисел. В результате возвращается остаток.

```
7 % 3 # -> 1
-7 % 3 # -> 2
7 % -3 # -> -2
```

string % object (форматирование)

Используется для форматирования объекта согласно правилам, заданным форматной строкой. В качестве объектов могут быть использованы числа, текст, индексный и ассоциативный массивы.

5. **number + number** (сложение)

   Используется для сложения двух чисел.

   ```
   1 + 3 # -> 4
   ```

   string + string (объединение текста)

   Используется для объединения двух текстов.

   ```
   "Ruby" + ?! # -> "Ruby!"
   ```

   array + array (объединение массивов)

   Используется для объединения элементов двух массивов.

   ```
   [1, 2] + [3, 4] # -> [1, 2, 3, 4]
   ```

   number - number (вычитание)

   Используется для вычисления разности двух чисел.

   ```
   2 - 1 # -> 1
   ```

   array - array (удаление элементов)

   Используется для удаления элементов из первого операнда.

   ```
   [1, 2, 2, ?R] - [2, ?1] # -> [1, "R"]
   ```

6. **number « integer; number » integer** (побитовый сдвиг)

   Сдвиг влево или вправо каждого бита на указанное количество разрядов.

   ```
   1 << 2 # -> 4
   ```

   string « string (добавление текста)

   Используется для добавления фрагмента в конец текста.

   Если вместо второго операнда передается целое число, то оно обрабатывается как кодовая позиция символа.

   ```
   "Ruby" << ?! # -> "Ruby!"
   "Ruby" << 33 # -> "Ruby!"
   ```

   array « object (добавление элемента)

   Используется для добавления элемента в конец массива.

   ```
   [1] << 2 # -> [ 1, 2 ]
   ```

7. **integer & integer** (побитовое И)

   Используется для сравнения битов.

   Если биты в одинаковых разрядах установлены в 1, то результирующий бит также устанавливается в 1.

   ```
   0b01 & 0b10 # -> 0
   ```

   array & array (пересечение множеств)

   Используется для получения элементов, содержащихся в обоих массивах.

   ```
   [1, 2, 2, 3] & [2, 3] # -> [2, 3]
   ```

8. **integer | integer** (побитовое ИЛИ)

   Используется для сравнения битов.

   Если любой из битов в одинаковых разрядах установлены в 1, то результирующий бит также устанавливается в 1.

   ```
   0b01 | 0b10 # -> 3
   ```

   array | array (объединение множеств)

   Используется для получения элементов, содержащихся хотя бы в одном из массивов.

   ```
   [1, 2, 2, 3] | [2, 3] # -> [1, 2, 3]
   ```

   integer ^ integer (побитовое исключающее ИЛИ)

   Используется для сравнения битов.

   Если один (и только один) из битов в одинаковых разрядах установлены в 1, то результирующий бит также устанавливается в 1.

   ```
   0b01 ^ 0b10 # -> 3
   ```

9. < <= >= > (отношение)

   Проверка отношения двух объектов.

   Для чисел:

```
1 <= 2 # -> true
1 > 2 # -> false
```

Для текста:

При проверке текста последовательно проверяется каждый байт. Первый отрицательный результат станет результатом выполнения всего выражения.

Каждая следующая буква алфавита считается больше, чем предшественница.

Любая строчная буква считается больше, чем любая прописная буква.

```
"а" < "б" # -> true
"а" < "Б" # -> false
"1" <= "2" # -> true
```

10. **object == object** (равенство)

    Проверка равенства двух объектов. Объекты считаются равными, если их значения и типы равны.

```
1 == 1.0 # -> true
1 == "1" # -> false
```

object != object (неравенство)

Проверка равенства двух объектов.

```
1 != 1.0 # -> false
1 != "1" # -> true
```

object === object (проверка условия)

Оператор используется для проверки условия. Чаще всего он применяется в составе предложений.

```
1 === 1.0 # -> true
1 === "1" # -> false

String  === 'matz' # -> true
Numeric === 42     # -> true
```

string =~ regexp; regexp =~ string (поиск совпадений)

Используется для поиска по тексту совпадений с образцом. В результате возвращается индекс символа, с которого начинается найденный фрагмент, или ссылка на nil, если совпадений не найдено.

string !~ regexp; regexp !~ string (отсутствие совпадений)

Проверка отсутствия совпадений.

object <=> object (сравнение)

Сравнение двух объектов.

```
        < = >
# ->    -1 0 1
```

-1 если первый операнд меньше второго;
0 если операнды равны;
1 если первый операнд больше второго;
nil если сравнение операндов невозможно (разные типы операндов).

Сравнить можно два числа, текста, индексных массива (последовательно сравнивается каждый элемент).

```
1 <=> 1.0 # -> 0
1 <=> ?2 # -> nil
```

11. **expression && expression** (логическое И)

    Второй операнд вычисляется только при положительном результате вычисления первого операнда.

    ```
    4 && 2 - 1 # -> 1
    3 > 4 && 2 - 1 # -> false
    ```

12. **expression || expression** (логическое ИЛИ)

    Второй операнд вычисляется только при отрицательном результате вычисления первого операнда.

    ```
    4 || 2 - 1 # -> 4
    3 > 4 || 2 - 1 # -> 1
    ```

13. **expression ? expression : expression** (логическое условие)

    В зависимости от результата вычисления первого операнда вычисляется второй операнд (для истинных значений) или третий операнд (для ложных значений)

    ```
    1 > 2 ? true : false # -> false
    1 < 2 ? true : false # -> true
    ```

14. **identificator = object** (присваивание)

    Используется для присваивания значений идентификаторам.

    ****= *= /= %= += -= «= »= &&= &= ||= |= ~=** (псевдооператоры)

    Псевдооператоры - это операторы, получившиеся в результате объединения с оператором присваивания.
    `<1> op= <2>` аналогично `<1> = <1> op <2>`

15. **not object** (логическое отрицание)

    Аналогично `!object`, но с меньшим приоритетом.

16. **expression and expression** (логическое И)

    Аналогично `expression && expression`, но с меньшим приоритетом.

    experssion or expression (логическое ИЛИ)

    Аналогично `expression || expression`, но с меньшим приоритетом.

4.2. Предложения

Предложения - это одна из разновидностей сложных выражений, создаваемая с помощью инструкций (поэтому количество возможных видов предложений строго ограничено). Каждое предложение начинается с инструкции, объявляющей тип предложения и заканчивается инструкцией end, объявляющей конец предложения. Между этими двумя инструкциями находится фрагмент кода, называемый телом предложения, ходом выполнения которого манипулирует предложение.

4.2.1. Условное предложение

Условные предложения управляют ходом выполнения в зависимости от логического значения переданного условия. В результате выполнения предложения возвращается результат выполнения последнего выражения в его теле (или nil).

Синтаксис предложения:

Тело предложения выполняется при положительном результате вычисления условия.

```
if условие
  тело_предложения
end
```

Тело предложения должно быть отделено от условия либо переводом строки, либо точкой с запятой, либо инструкцией then.

Инструкция else:

Содержит фрагмент кода, который выполняется при отрицательном результате вычисления условия.

```
...
  тело_предложения
else
  код
end
```

При необходимости записать подряд инструкцию else и новое условное предложение используется инструкция elsif.

Краткий синтаксис:

```
тело_предложения if условие
```

Тело предложения либо должно находиться на одной строке с условием, либо быть ограничено инструкциями begin и end, либо быть ограничено двумя круглыми скобками.

```
begin
  тело_предложения
end if условие
```

или

```
( тело_предложения
) if условие
```

Если вместо инструкции if использовать инструкцию unless, то тело предложения будет выполняться при отрицательном результате вычисления условия. Инструкция elsif в этом случае не используется.

4.2.2. Разветвленное условие

Разветвленные условия похожи на условные предложения, но позволяют проверять сразу несколько условий. Условия проверяются последовательно. Обычно наиболее вероятные варианты записывают раньше остальных - это уменьшит время выполнения предложения. В результате выполнения предложения возвращается результат выполнения последнего выражения в его теле (или nil).

Синтаксис предложения:

Выполняется фрагмент кода с положительным результатом вычисления условия.

```
case
  when условие
    код
  when условие
    код
  ...
end
```

Условие может состоять из нескольких выражений, разделенных запятыми. Любое из них может послужить причиной для выполнения кода.

Специальный синтаксис:

Проверяется равенство значения условия и указанных выражений (с помощью ===).

```
case условие
when выражение
  код
when выражение
  код
...
end
```

В любой форме можно использовать инструкцию else.

4.2.3. Цикл

Цикл - это предложение итеративного типа (заставляющее программу повторно выполнять некоторый фрагмент кода). Циклы используются в том случае, когда число итераций неизвестно заранее. В результате выполнения цикла возвращается ссылка на nil.

Цикл while не соответствует слову "пока" в естественном языке. Условие цикла не проверяется непрерывно, а только до или после каждой итерации.

Синтаксис предложения:

Тело предложения выполняется до тех пор пока результат вычисления условия истинен (условие проверяется до итерации).

```
while условие
  тело_цикла
end
```

Тело цикла должно быть отделено от условия либо переводом строки, либо точкой с запятой, либо инструкцией do.

Сокращенный синтаксис:

```
тело_цикла while условие
```

Тело цикла и условие обязательно должны находиться на одной строке кода. Если тело цикла ограничено инструкциями begin и end, то условие проверяется после итерации. Также тело цикла может быть ограничено двумя круглыми скобками.

```
begin
  тело_цикла
end while условие
```

или

```
( тело_цикла
) while условие
```

Если вместо инструкции while использовать инструкцию until, то тело цикла будет выполняться до тех пор пока результат вычисления условия ложен.

Использование инструкции until эквивалентно конструкции while not.

Для создания бесконечного цикла существует частный метод экземпляров из модуля Kernel - `.loop { }`, который используется для бесконечного выполнения блока до возникновения исключения `StopIteration` или вызова специальных инструкций. Данный метод эквивалентен конструкции while true.

4.2.4. Перебор элементов

Перебор элементов - это предложение итеративного типа, выполняющее фрагмент кода для каждого элемента составного объекта.

```
for параметр in объект
  тело_перебора
end
```

Параметр в теле перебора ссылается на элементы составного объекта (может быть использовано несколько параметров, разделенных запятыми).

Для составного объекта должен существовать метод `.each`, который будет использоваться в ходе выполнения предложения.

Тело предложения должно быть отделено либо переводом строки, либо точкой с запятой, либо инструкцией do.

4.2.5. Управление ходом выполнения

Ход выполнения предложения может быть изменен с помощью специальных инструкций в его теле. Необязательный код, передаваемый инструкции возвращается в результате выполнения предложения. Если код отсутствует, то возвращается ссылка на nil.

Инструкции:

return [код] - используется для завершения выполнения предложения и всех методов, в теле которых оно выполняется, продвигаясь вверх по областям видимости;

break [код] - используется для завершения выполнения предложения;

next [код] - используется для завершения текущей итерации цикла и начала следующей;

redo - используется для повторного выполнения текущей итерации. Проверка условия при этом не выполняется.

4.3. Триггеры

В качестве условия могут быть использованы триггеры. Триггер - это сложное выражение, составленное с помощью операторов .. или .... Эти операторы имеют приоритет выполнения больше, чем у оператора условия и меньше, чем у оператора логического ИЛИ (примерно 12.5). В условии может использоваться только один триггер.

Как и обычные условия, триггеры имеют некоторое логическое значение. Его особенность в том, что оно может изменяться в зависимости от результатов предыдущих вычислений, т.е. триггер сохраняет состояние выполнения.

Обычно триггеры применяются вместе с регулярными выражениями для обработки текста между начальными и конечными шаблонами.

Синтаксис триггера:

```
условие..условие; условие...условие
```

Операнды триггера - это два условия, при проверке которых логическое значение триггера либо остается неизменным, либо изменяется на противоположное. Порядок проверки условий зависит от оператора, используемого для создания триггера.

Значение триггера ложно до тех пор пока ложно значение первого условия. После смены логического значения первого условия изменяется и логическое значение триггера. Оно будет сохраняться до тех пор пока логическое значение второго условия ложно. После смены логического значения второго условия, изменяется и логическое значение триггера и проверка условий начинается сначала.

Процесс выполнения триггера:

1. Если логическое значение триггера ложно, то проверяется первое условие. В зависимости от его логического значения устанавливает логическое значение триггера. После его возвращения для оператора . . также проверяется второе условие.
2. Если логическое значение триггера истинно, то проверяется второе условие. Если логическое значение второго условия истинно, то логическое значение триггера меняется.

По смыслу триггеры довольно похожи на диапазоны. Они верны до тех пор пока существующее состояние выполнения находится от достижения первого условия и до достижения второго условия. Оператор '...' включает состояние достигнувшее второе условие, а оператор '..' - нет.

Глава 5

Реализация ООП

Одна из главных особенностей Ruby - сильно выраженная поддержка объектно-ориентированной парадигмы.

- Любые данные, в том числе и элементарные, относятся к объектам.
- Большинство операторов относится к методам.
- Множество синтаксического сахара облегчает использование основных технологий и сущностей ООП.

Основные особенности:

- Любые данные хранятся в виде объектов;
- Вычисления выполняются путем взаимодействия (обмена данными) между объектами, при котором один из объектов требует выполнение некоторого действия от другого. Объекты взаимодействуют с помощью методов. Метод - это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при его выполнении;
- Каждый объект имеет независимую память, которая состоит из других объектов;
- Каждый объект является представителем класса, который определяет общие свойства объектов;
- В классе также определяется поведение объекта - набор доступных методов. Все экземпляры класса могут выполнять одни и те же действия;
- Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение экземпляров базового класса автоматически доступны в производном классе.

5.1. Основные сущности

5.1.1. Модули (Module)

Модули - это абстрактные сущности, использующиеся для инкапсуляции методов, констант или фрагмента программы.

В качестве пространств имен модули обычно используются для инкапсуляции области видимости программы. При этом все классы или модули программы объявляются только в теле одного отдельного модуля. Это позволяет разным программам не засорять глобальную область видимости. Использование глобальной области видимости программой считается плохим тоном среди программистов, т.к. требует согласования использованных имен классов и модулей и может привести к непредсказуемым последствиям.

Явно выраженный пример инкапсуляции логически связанных фрагментов кода представляет модуль Math, объединяющий математические функции.

С точки зрения синтаксиса модуль - это фрагмент кода, связанный с константой. В отличии от классов модули не могут иметь экземпляров или производных, поэтому любой класс может считаться модулем, но не любой модуль - считаться классом.

Создание модуля называется объявлением, а заполнение области видимости модуля - определением.

```
module идентификатор_модуля
  тело_модуля
end
```

- Инструкция module ожидает получения константы, сообщая интерпретатору область памяти в которую необходимо выполнить запись информации. Если соответствующей области не существует, то автоматически объявляется новый модуль. В другом случае будет изменен уже существующий модуль;
- В теле модуля существует отдельная область видимости. В ней могут создаваться любые другие сущности. Внутри тела модуля псевдопеременная self ссылается на модуль. В результате создания модуля возвращается результат выполнения последнего выражения в теле (обычно это nil).

Модули также относятся к экземплярам класса Module. Методы экземпляров, определенные в этом классе, могут вызываться для всех модулей.

::new { nil } *# -> module [Module]*

Используется для создания анонимного модуля. Необязательный блок выполняется в теле модуля. Модуль перестанет быть анонимным, если будет присвоен константе.

5.1.2. Классы (Class)

Класс - это абстрактная сущность, расширяющая поведение модулей, применяемая для описания структуры и поведения ее экземпляров. Создание класса называется объявлением, а описание его экземпляров - определением. Любой класс (даже встроенный) может быть переопределен в любом месте кода.

```
class идентификатор_класса
  тело_класса
end
```

- Инструкция class ожидает получения константы, сообщая интерпретатору область памяти в которую необходимо выполнить запись информации. Если соответствующей области не существует, то автоматически объявляется новый класс. В другом случае будет изменен уже существующий класс.
- В теле класса создается отдельная область видимости. В ней может определяться поведение экземпляров класса или любые другие сущности. Внутри тела класса псевдопеременная self ссылается на класс. В результате создания класса возвращается результат выполнения последнего выражения в теле класса (обычно это nil).

Классы также относятся к экземплярам класса Class. Методы экземпляров, определенные в этом классе, могут вызываться для всех остальных классов.

::new(class = Object) *# -> class [Class]*

Используется для создания анонимного класса, наследующего переданному. Класс перестанет быть анонимным, если будет присвоен константе.

Экземпляры классов: объекты, создаваемые с помощью классов.

Методы, используемые для создания объекта, называются конструкторами. По умолчанию в Ruby для каждого класса определен конструктор `::new`. Этот метод выполняет два действия: создает область памяти для объекта (метод `.allocate`) и заполняет эту область (метод `.initialize`).

Метод `.initialize` должен определяться разработчиком. Определяемый метод автоматически становится частным. Инициализируемые в нем переменные экземпляра доступны для любого метода создаваемого объекта.

Метод .allocate уже определен в классе Class и его переопределение обычно не требуется.

Собственный класс объекта: отдельный класс для каждого объекта.

Каждый объект кроме класса, к экземплярам которого относится, имеет еще и свой собственный класс. С помощью собственных классов для объектов определяется уникальное поведение. Собственные классы также называют метаклассами (metaclass).

```
class << объект
  тело_класса
end
```

Это предложение определяет собственный класс для переданного объекта (объект рационально передавать с помощью переменной). В теле класса в этом случае используется псевдопеременная self.

Возвращается результат выполнения последнего выражения в теле собственного класса объекта (обычно это nil).

Следует запомнить, что классы не наследуют методы своих собственных классов, а относятся к их экземплярам.

5.1.3. Константы

Константы определяются в теле модуля и служат для хранения известных и не изменяющихся данных, общих для всего класса.

Использовать константу можно с помощью бинарного оператора `::`, имеющего наивысший приоритет и последовательность выполнения L. В качестве левого операнда используется идентификатор модуля, а в качестве правого операнда - константа. Использование константы называют ее вызовом.

```
класс::константа
```

Использование этого выражения с оператором присваивания `=`, позволяет изменить константу в любом месте кода.

Для того, чтобы вызвать константу, интерпретатор должен определить какая именно константа должна быть использована. При полной записи выражения поиск осуществляется в указанном классе. При отсутствии левого операнда по умолчанию используется псевдопеременная self.

При наличии только константы, поиск будет выполняться в соответствии с иерархией областей видимости.

1. В теле класса, в котором была вызвана константа;
2. В теле модуля, в котором определяется класс;
3. В теле модуля, добавленного к классу;
4. Вверх по иерархии областей видимости с выполнением пунктов 1-3;
5. Вызов метода `object.const_missing` для класса, в теле которого была вызвана константа.

5.1.4. Переменные

Переменные определяются в теле класса и служат для хранения изменяющихся данных: состояния класса и его экземпляров. Переменные могут существовать в двух различных областях видимости: области видимости класса и области видимости экземпляра класса.

Переменные класса: переменные, объявляемые в области видимости класса.

Переменные класса могут быть объявлены в любом месте тела класса (в том числе и в теле метода). Лексема переменной начинается с символов @@.

Переменная класса может быть использована в любом месте тела класса (в том числе и в теле метода). Она ссылается на один и тот же объект для всех его экземпляров.

Переменные класса используются для хранения изменяющихся данных, общих для всех экземпляров класса. Обычно переменные класса инициализируются в теле класса и используются в теле его методов. Использование несуществующей переменной класса считается ошибкой.

Переменные экземпляра: переменные, объявляемые в области видимости экземпляра класса.

Переменные экземпляра могут быть объявлены в теле метода экземпляров. Лексема переменной начинается с символа @.

Переменная экземпляра может быть использована в теле любого метода экземпляров. Она ссылается на разные объекты для каждого отдельного экземпляра.

Переменные экземпляра используются для хранения состояния экземпляров класса. Обычно переменные экземпляра инициализируются в теле метода в момент его вызова. Использование несуществующей переменной экземпляра считается её объявлением.

Если определить переменную экземпляра в теле класса, а не в теле метода, то она будет связана с классом, а не с его экземплярами. Такие переменные недоступны в теле методов и фактически относятся к переменным экземпляра объекта, принадлежащего к классу Class.

5.1.5. Методы

Методы - это абстрактные сущности, определяющие действия, которые может выполнять объект, и действия, которые можно выполнять с самим объектом (например, изменять его состояние). Выполняемые действия делятся на два вида: функции и процедуры. В Ruby их четкого синтаксического разграничения не существует - процедуры входят в подмножество функций. С точки зрения языка программирования, метод относится к выражениям.

- *Функция* - это метод, используемый для получения объекта (например математические функции);

  Процедура - это метод, используемый для выполнения действия (например сохранение данных в файл). В результате выполнения процедуры обычно возвращается статус выполнения (логическая величина) или ссылка на объект, для которого вызывалась процедура.

С точки зрения синтаксиса, метод - это именованный фрагмент кода, выполняющий одну конкретную задачу. Методы определяются в теле класса. В результате определения возвращается nil. Каждый метод связан с тем объектом, для которого был определен и не может использоваться без ссылки на него. Использование метода также называют вызовом.

Определение метода

Методы экземпляров класса: методы, определяющие поведение экземпляров класса.

```
def идентификатор_метода(параметры)
  тело_метода
end
```

Хранение методов экземпляров - это главное отличие классов от остальных сущностей.

Методы класса: методы, определяющие поведение класса.

```
def класс.идентификатор_метода(параметры)
  тело_метода
end
```

Вместо идентификатора класса может использоваться псевдопеременная self. Это повышает переносимость кода, определяя метод для любого текущего класса, а не для какого-то конкретного.

Методы класса не являются отдельной сущностью - это всего лишь методы, создаваемые в теле собственного класса объекта (объектом в этом случае считается класс).

Класс:

- методы экземпляров хранятся в теле класса;
- методы класса (собственные методы) хранятся в теле собственного класса объекта (объектом в данном случае является класс).

Следовательно главное отличие классов от обычных объектов в возможности хранить методы своих экземпляров.

Синтаксис метода:

Идентификатор метода: лексема идентификатора метода аналогична лексеме локальной переменной.

Обычно имя метода выбирается в соответствии с его целью:

- Для именования процедур используются глаголы;
- Имя функции обычно описывает объект, который она возвращает;
- Имя методов с побочным эффектом заканчивается восклицательным знаком (добавляется только если существует версия метода без побочного эффекта);
- Предикаты - это методы, утверждающие или отрицающие что-либо об объекте. В результате выполнения предиката возвращается логическая величина, характеризующая истинность или ложность утверждения. Имя предиката обычно заканчивается вопросительным знаком.

Параметры: локальные переменные, которые служат для разделения аргументов (объектов, передаваемых методу при вызове).

Несколько параметров отделяются запятыми. При вызове метода параметры будут последовательно инициализироваться переданными аргументами с помощью выражения присваивания.

- Когда параметр инициализируется при объявлении метода, то он имеет значение по умолчанию. Это значение будет использоваться, если при вызове метода необходимый аргумент не передавался. Значения по умолчанию могут быть произвольными выражениями, переменными экземпляра или даже другими параметрами, объявленными ранее. Параметры, имеющие значение по умолчанию, должны объявляться последовательно;
- Когда перед параметром используется оператор разыменования (`*`), то такой параметр принимает произвольное количество аргументов, сохраняемых в массиве. Подобный тип параметров объявляется после параметров, имеющих значение по умолчанию. Допускается существование только одного такого параметра для метода;
- Когда перед последним параметром используется амперсанд (`&`), то такой параметр принимает блок (вызывая для аргумента метод `.to_proc`). В теле метода может использоваться как параметр, так и инструкция yield.

Для повышения читабельности параметров, ссылающихся на логическую величину может использоваться приставка is.

Тело метода: фрагмент кода, выполняемый в момент вызова метода.

Тело метода использует область видимости объекта. В ней доступны объявленные параметры и переменные экземпляра. В теле метода псевдопеременная self ссылается на объект, для которого метод был вызван.

Синонимы: копия метода, имеющая другой идентификатор.

Для метода может быть определено любое количество синонимов. Синонимы могут быть созданы только в теле класса, объявляющего метод.

Создание синонима выполняется с помощью инструкции alias: `alias синоним идентификатор_метода`

Запомнить синтаксис выражения будет проще, если рассматривать его относительно выражения присваивания: синоним = метод.

Удаление метода: выполняется с помощью инструкции undef. Ее можно использовать только в теле класса, определяющего данный метод или в теле его подклассов (в этом случае метод будет удален только для отдельного подкласса).
`undef идентификатор_метода`

Ruby 2.0

Раньше для передачи именованных аргументов использовались ассоциативные массивы. Теперь для этого могут объявляться именованные параметры. Они существуют во многих языках, а теперь добавлены и в Ruby.

Именованные параметры - это синтаксический сахар, облегчающий идентификацию получаемых аргументов.

- Когда после параметра используется двоеточие, то он считается именованным. Значение после двоеточия будет считаться значением по умолчанию.

```
def method(name: default); name; end
method(name: 'method') # -> method
```

Передача необъявленных именованных аргументов считается исключением.

```
def method(name: default); name; end
method(args: 'method') # -> ArgumentError!
```

- Когда перед параметром используется разыменование (**), то такой параметр принимает произвольное количество именованных аргументов, сохраняемых в ассоциативном массиве. Имя параметра указывать обязательно (нельзя использовать только **)

```
def method(**options); options; end
method(name: 'method') # -> { name: 'method'}
```

Блоки

Блоки - одна из особенных синтаксических конструкций в Ruby, предоставляющая мощные возможности по выполнению кода.

Блоки - это один из вариантов создания замыканий. Они являются частью окружающего кода и могут использовать объявленные переменные.

С точки зрения синтаксиса, блок - это фрагмент кода, связанный с группой параметров. Блоки не могут использоваться сами по себе, а только передаваться методам. Блок всегда должен передаваться методу последним. Если объект не использует блок, то его передача игнорируется.

```
<method> { |параметры| тело_блока }
```

или

```
<method> do |параметры|
  тело_блока
end
```

Открывающая фигурная скобка или инструкция do относятся к предыдущему выражению - остальные аргументы необходимо ограничивать круглыми скобками, иначе блок будет передан последнему аргументу.

Методы, позволяющие перебирать элементы составного объекта, не зависимо от их типа, называются итераторами. Перебор элементов выполняется с помощью блока, которому они передаются. Каждое отдельное выполнение блока называется итерацией. В результате выполнения итератора обычно возвращается ссылка на объект, для которого он был вызван.

Вместо блока итераторы также могут принимать идентификатор метода, начинающийся с амперсанда. В этом случае метод будет вызван для каждого элемента составного объекта.

Итераторы в Ruby - это синтаксический сахар, использующийся вместо предложения перебора (for in).

Синтаксис блока:

Параметры: локальные переменные, которые служат для разделения аргументов (объектов, передаваемых блоку).

Несколько параметров отделяются запятыми. При выполнении блока параметры будут последовательно инициализироваться переданными аргументами с помощью выражения присваивания.

Параметры блока не могут иметь значений по умолчанию или принимать блоки. Однако они могут ссылаться на массив аргументов (с помощью оператора разыменования `*`).

Тело блока: создает собственную область видимости. В нем определены объявленные в окружающем коде переменные, параметры и переменные экземпляра. В теле блока псевдопеременная self ссылается на объект, для которого был вызван метод, принимающий блок.

Блок относится к замыканиям - в теле блока существуют локальные переменные, объявленные в окружающем коде. Чтобы явно указать переопределение локальных переменных, их идентификаторы отделяют от параметров блока точкой с запятой (`| x, y; z, k, n |`). Переопределенные локальные переменные в теле блока будут ссылаться на nil.

Выполнение блока: блок выполняется в момент вызова. Возвращается результат выполнения последнего выражения.

Для вызова блока в теле метода используется инструкция yield, которой передаются аргументы, отправляемые в блок (нельзя передавать блоки). Избыток аргументов игнорируется. Возвращается результат выполнения блока.

Использование инструкции yield при отсутствии блока считается исключением `LocalJumpError`.

Присваивание аргументов параметрам выполняется так же, как и выражение присваивания.

```
hash = { a: [], b: [] }
hash.each { |key, (first_in_value, second_in_value)| }
```

Ход выполнения блока может регулироваться с помощью специальных инструкций в его теле. Необязательный код, передаваемый инструкции становится результатом выполнения (по умолчанию nil).

Инструкции:

return [код] - используется для завершения выполнения блока и всех методов, в теле которых оно выполняется, продвигаясь вверх по областям видимости;

break [код] - используется для завершения выполнения блока и принимающего блок метода;

next [код] - используется для завершения выполнения блока. Итератор может начать новую итерацию;

redo - используется для повторного выполнения блока.

Вызов метода

Методы вызываются с помощью бинарного оператора . (или ::, но он обычно не используется или используется только для вызова методов класса), имеющего наивысший приоритет и последовательность выполнения L. Левым операндом должен быть объект, для которого вызывается метод (получатель сообщения), а правым операндом - идентификатор метода.

```
объект.идентификатор_метода(аргументы)
```

Аргументы - это группа объектов, инициализирующих параметры. Несколько объектов разделяются запятыми. Недостаток или избыток аргументов считается исключением. Количество требуемых аргументов соответствует количеству объявленных параметров.

Возвращается результат выполнения последнего выражения в его теле. Вызов метода без указания объекта вызывает его для псевдопеременной self.

Аргументы классифицируют по способу их передачи.

- Передача по значению - метод копирует переданный аргумент. Изменение параметра не влияет на передаваемый аргумент;
- Передача по адресу - передаваемым значением является адрес, по которому можно найти значение переменной;
- Передача по ссылке - метод копирует не переданный аргумент, а его адрес, однако использует синтаксис, при котором работа выполняется с объектом, хранящимся по этому адресу. Изменение параметра также изменит переданный аргумент.

Автор книги разделяет точку зрения, что в Ruby все аргументы передаются по ссылке. Это отражает суть происходящего при изменении состояния передаваемых аргументов, но читателям также стоит обратить внимание на то, что передача аргументов методам, имеет свою специфику, связанную в основном с выражением присваивания и тонкостями терминологии в контексте языков высокого уровня.

Синтаксис вызова:

- использование круглых скобок при вызове метода не обязательно (кроме случаев, когда методу передается блок);
- когда последним аргументом метода передается ссылка на ассоциативный массив, то использование фигурных скобок для выделения массива не требуется;

- для извлечения элементов составного объекта используется оператор разыменования (`*`). Элементы извлекаются с помощью метода `.to_splat`;
- для передачи блока, аргумент должен передаваться последним и начинаться с амперсанда (`&`).

Перед вызовом метода интерпретатор должен определить какой именно метод использовать. Поиск метода осуществляется в соответствии с иерархией наследования класса объекта (учитывается и собственный класс объекта).

- в собственном классе объекта;
- в теле класса объекта;
- в теле модуля, добавленного к классу;
- вверх по иерархии классов с выполнением пунктов 2 и 3;
- вызов для объекта метода `.method_missing`.

Ruby 2.0

- для извлечения элементов ассоциативного массива используется оператор разыменования (`**`). Несколько ассоциативных массивов подряд будут объединяться.

```
APP_OPTS = { name: 'play', author: 'dave' }
LOG_OPTS = { level: 2, color: 'blue', line: '3pt' }

def log(msg, options)
p msg
p options
end

log( 'Starting', **APP_OPTS )
# ->
# 'Starting'
# { name: 'play', author: 'dave' }

log( 'Connected', **APP_OPTS, **LOG_OPTS )
# ->
# 'Connected'
# { name: 'play', author: 'dave',
#   level: 2, color: 'blue', line: '3pt' }

log( 'Giving up', **APP_OPTS, **LOG_OPTS, color: 'VERY RED' )
# ->
# 'Giving up'
# { name: 'play', author: 'dave',
#   level: 2, color: 'VERY RED', line: '3pt' }
```

Ход выполнения

Ход выполнения метода может регулироваться с помощью специальных инструкций в его теле. Необязательный код, передаваемый инструкции становится результатом выполнения (по умолчанию возвращается nil).

Инструкции:

return [код] - используется для завершения выполнения метода;
break [код] - создание исключения;
next [код] - создание исключения;
redo - создание исключения.

Собственный метод объекта: метод, определенный в собственном классе объекта.

Собственные методы объекта могут быть определены либо в теле собственного класса (как методы экземпляров), либо с помощью отдельного предложения.

```
def объект.метод
  тело_метода
end
```

Определение такого метода возможно только после создания объекта (объект также должен существовать в текущей области видимости). Очевидно, что объект должен быть представлен в виде переменной или константы.

В результате возвращается nil.

Поведение всего класса зависит от набора существующих методов класса. Поэтому, неудивительно, что методы класса также относятся к собственным методам объекта (которым в данном случае является класс).

5.1.6. Примеры

Определение простого класса в теле модуля.

Метод `.to_s` используется при отображении экземпляра класса.

```
module OurKlass
  class User
    @@count = 0 # class var.

    # class method.
    def self.count
      @@count
    end

    def initialize( name, age )
      @name = name # instance var.
      @age = age
      @@count += 1
      @id += 1
    end

    # instance method.
    def to_s
      "#{@name}: #{@age} years old"
    end

  end # class User.
end # module.
```

Создание экземпляра класса. Метод `.to_a` еще не определен.

```
user = OurKlass::User.new "Timmy", 22 # -> "Timmy: 22 years old"
OurKlass::User.count # -> 1
user.to_a # -> error!
```

Определение метода `.to_a`.

```
class OurKlass::User
  def to_a
    [ @name, @age ]
  end
end

user.to_a # -> ["Timmy", 22]
```

Различные способы определения метода класса.

```
class OurKlass::User
  def self.next_count
    @@count + 1
  end
end

OurKlass::User.next_count # -> 2

class OurKlass::User
  class << self
    def prepend_count
      @@count - 1
    end
  end
end

OurKlass::User.prepend_count # -> 0
```

Определение собственного метода объекта. При переопределении переменной для нового объекта метод определен уже не будет.

```
class << user
  def id
    @id
  end
end

user.id # -> 1
user = OurKlass::User.new "Tommy", 33 # -> "Tommy: 33 years old"
user.id # -> error!
```

5.2. Основные принципы

5.2.1. Инкапсуляция

Инкапсуляция состояния

По умолчанию состояние объекта доступно только внутри объекта. Поэтому для получения или изменения значения переменных экземпляра или класса требуется явно определять соответствующие методы, также называемые свойствами. Свойства позволяют получать состояние объекта (читать значение переменной) изменять состояние объекта (определять значение переменной).

Свойства для получения состояния обычно просто возвращают текущее значение переменой. Для этого в конце тела метода, последним выражением должен быть идентификатор необходимой переменной.

Для изменения состояния в конце тела метода, последним выражением должно быть выражение присваивания, в котором участвуют идентификатор требуемой переменной и переданный методу аргумент. Идентификатор свойства при этом обычно заканчивается знаком равенства (`<attr>=`).

Для объявления свойств могут использовать частные методы экземпляров из класса Module.

.attr_accessor(*attribute) *# -> nil*

Используется для объявления переменной экземпляра и свойств для получения и изменения её значения.

.attr_reader(*attribute) *# -> nil*

Используется для объявления переменной экземпляра и свойства для получения её значения.

.attr_writer(*attribute) *# -> nil*

Используется для объявления переменной экземпляра и свойства для изменения её значения.

Инкапсуляция поведения

Инкапсуляция поведения осуществляется посредством ограничения доступа к методам.

Методы могут быть как внутренними (обеспечивающими логику функционирования объекта), так и внешними (используемыми для взаимодействия объектов). Доступ к внутренним методам может быть ограничен областью видимости класса.

Классификация методов:

- *public* - общие методы. Общие методы позволяют объектам взаимодействовать друг с другом. Они могут быть вызваны в любой области видимости;

  protected - защищенные методы. Защищенные методы позволяют объектам одного типа взаимодействовать друг с другом. Они могут быть вызваны только в области видимости класса (и его производных) или экземпляров класса (и его производных) и только для экземпляров того же класса (и его подклассов);

  private - частные методы. Частные методы реализуют внутреннюю логику объекта. Они могут быть вызваны только в области видимости класса (и его производных) или экземпляров класса (и его производных) и только для текущего экземпляра (частные методы всегда вызываются для псевдопеременной self). Частные методы помогают скрывать реализацию работы программы и разрешить доступ только к API.

Частные методы реализуют инкапсуляцию поведения объекта, а защищенные - инкапсуляцию поведения класса объектов.

Область применения метода объявляется с помощью частных методов экземпляров из класса Module: `.public`, `.private` и `.protected`. Ограничивается применение указанных методов или методов, объявляемых после. Методы, определяемые вне тела класса, относятся к частным методам класса Object.

.private_class_method(*name) *# -> self [Module]*

Используется для объявления методов класса частными. Обычно применяется для инкапсуляции конструкторов.

.public_class_method(*name) *# -> self [Module]*

Используется для объявления методов класса общими.

.module_function(*name) *# -> self [PRIVATE: Module]*

Используется для объявления методов экземпляров частными и создания аналогичных методов модуля. Это позволяет использовать методы либо непосредственно с помощью модуля, либо добавляя их к уже существующему классу.

При вызове без аргументов влияет на все методы, объявленные далее.

5.2.2. Наследование и агрегация

Наследование

Реализация наследования в Ruby позволяет создавать производные классы (использовать функциональность уже существующих классов).

В Ruby реализовано единичное наследование. Это означает, что любой класс может иметь только один базовый класс.

Наследование класса может быть выражено словосочетанием ”относится к” или ”принадлежит к”.

```
class производный_класс < базовый_класс
```

- Переменные экземпляра не наследуются;
- Переменные класса наследуются его производными. При этом они ссылаются на тот же объект, что и в базовом классе;
- Константы наследуются производными классами. При этом они ссылаются на разные объекты для каждого подкласса;
- Методы наследуются производными классами. При этом для каждого класса существует своя копия метода.

Агрегация

Агрегация - это технология языка, позволяющая сущности использовать структуру другой сущности путем включения (одна сущность может содержать ссылку на другие сущности). При удалении основной сущности, вложенные сущности продолжают существовать.

Реализация агрегации в Ruby позволяет добавлять функциональность модулей к объектам и их классам.

В Ruby реализована множественная агрегация. Это означает, что любой объект может включать произвольное количество модулей.

Включение модуля может быть выражено словом ”содержит”.

.include(*a_module) *# -> self [PRIVATE: Module]*

Используется для агрегации модулей. Агрегируемые модули добавляются в начало иерархии наследования. Равносильно копированию методов из модуля в класс объекта.

.extend(*a_module) *# -> object [Object]*

Используется для агрегации модулей. Агрегируемые модули добавляются в начало иерархии наследования собственного класса объекта. Равносильно копированию методов из модуля в собственный класс объекта.

.extend_object(object) *# -> object [PRIVATE]*

Используется для включения текущего объекта в состав переданного аргумента.

```
object.extend self
```

Выполнение выражения ‘extend self‘ в теле класса приведет к тому, что для каждого метода экземпляров автоматически объявляется соответствующий метод класса.

Ruby 2.0

Во второй версии Ruby добавлена возможность переопределять методы, определенные в классе.

.prepend(*a_module) *# -> self [PRIVATE: Module]*

Используется для агрегации модулей. Агрегируемые модули добавляются в начало очереди вызова методов. Также переопределяются константы и переменные. Методы, определенные в самом классе будут доступны с помощью super.

```
module Foo
  def baz; 'foo-baz'; end
end

class Bat
  include Foo
  def baz; 'bat-baz'; end
end
Bar.new.baz # -> 'bat-baz'

class Bar
  prepend Foo
  def baz; 'bar-baz'; end
end
Bar.new.baz # -> 'foo-baz'
```

Этот механизм облегчает полиморфизм методов. Если раньше приходилось расширять методы посредством создания промежуточных синонимов, то теперь можно просто агрегировать различные модули.

```
# Ruby 1.9:
class Range
  # Взято из active_support/core_ext/range/include_range.rb
  # Изменение Range#include? для поиска диапазонов.
  def include_with_range?(value)
    if value.is_a?(::Range)
      # 1...10 включает 1..9, но не 1..10.
      operator = exclude_end? && !value.exclude_end? ? :< : :<=
      include_without_range?(value.first) && value.last.send(operator, last)
    else
      include_without_range?(value)
    end
  end

  alias_method_chain :include?, :range
end

Range.ancestors # -> [ Range, Enumerable, Object... ]

# Ruby 2.0
module IncludeRangeExt
  # Изменение Range#include? для поиска диапазонов.
  def include?(value)
    if value.is_a?(::Range)
      # 1...10 включает 1..9, но не 1..10.
      operator = exclude_end? && !value.exclude_end? ? :< : :<=
      super(value.first) && value.last.send(operator, last)
    else
      super
    end
  end
end

class Range
  prepend IncludeRangeExt
end

Range.ancestors # -> [ IncludeRangeExt, Range, Enumerable, Object... ]
```

Иерархия наследования

Иерархия классов объекта - это последовательность классов, в которых выполняется поиск вызываемого метода (в Ruby иерархия классов содержит также и модули).

- На вершине иерархии находится класс BasicObject, от него наследует класс Object, в теле которого выполняется программа.
- Класс Object включает модуль Kernel, в котором определено большинство основных методов.
- Любой модуль относится к экземплярам класса Module, который наследует класс Object. Методы экземпляров из класса Module могут вызываться в теле модулей.
- Любой класс относится к экземплярам класса Class, который наследует класс Module. Методы экземпляров из класса Class могут вызываться в теле классов.
- Методы экземпляров из класса Module могут вызываться в теле классов. Методы экземпляров из класса Class не могут вызываться в теле модулей.

Иерархия классов может изменяться в ходе выполнения программы.

- базовый класс и его иерархия добавляются в начало иерархии для экземпляров производного класса;
- собственный класс всегда находится в начале иерархии;
- включение модуля добавляет его в начало иерархии;

5.2.3. Полиморфизм

Реализация полиморфизма в Ruby позволяет переопределять унаследованные методы.

Виртуальные методы: методы, которые могут быть переопределены производным классом, так что конкретная реализация метода будет вычислена во время выполнения. В Ruby все методы относятся к виртуальным. Это значит что программист не обязан знать точный тип объекта, если у него имеется набор виртуальных методов - достаточно будет информации о принадлежности к требуемому классу.

Для переопределения метода, в теле класса, объявляют метод с тем же идентификатором.

Чтобы в теле унаследованного метода использовать метод базового класса вызывают инструкцию super, которой передаются необходимые аргументы (по умолчанию передаются все аргументы).

Абстрактный метод: метод, который был объявлен, но не определен. В Ruby все абстрактные методы считаются определенными и возвращают ссылку на nil.

Абстрактный класс: класс, содержащий хотя бы один виртуальный метод. Абстрактные классы используются только в иерархии наследования и не предназначены для создания экземпляров. Можно сказать, что абстрактные классы - это прототипы для создания других классов.

В Ruby создание экземпляров абстрактных классов по умолчанию не ограничивается, и фактически абстрактные классы не отличаются от обычных.

Класс Numeric, к которому относятся все числа - типичный абстрактный класс, а класс IO, к которому относятся файлы и потоки, абстрактным не является. В любом случае синтаксис определения этих классов не отличается.

Интерфейс: обязательства (контракт), которые берет на себя класс. Интерфейсы описывают функциональность, предоставляемую классом, реализующим интерфейс. Класс, реализующий интерфейс, должен определять все его методы. Один класс может реализовывать несколько интерфейсов одновременно.

Обычно интерфейсы реализуются как модули, содержащие только абстрактные методы.

```
module Openable
  def open; end
  def close; end
end
```

Переопределение операторов

Одной из полезных особенностей в Ruby является то, что большинство операторов на самом деле относятся к методам. Поэтому поведение операторов для разных типов объектов отличается - оно зависит от определения метода в теле класса. Также разрешается определять собственное поведение для различных операторов и переопределять уже существующее. Выражение, составленное с помощью операторов, при этом аналогично вызову метода с тем же идентификатором.

```
<op> object <-> object.<op>
first_object <op> second_object <-> first_object.<op>(second_object)
```

Операторы, не относящиеся к методам:

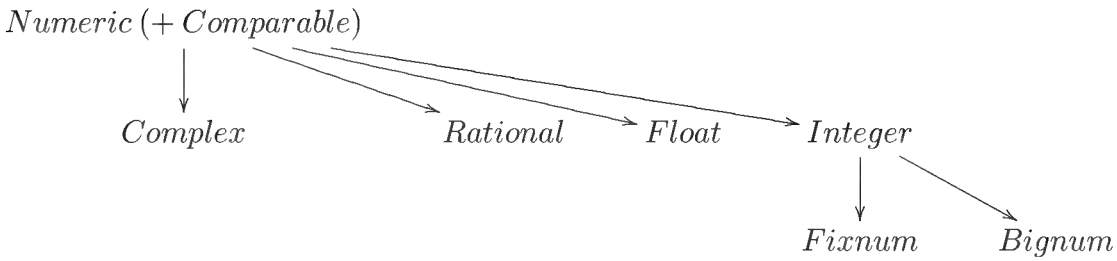
```
., ::, &&, ||, ?:, =, псевдооператоры, not, and, or, .., ...
```

Часть II

Описание классов

Глава 6

Числа



Для математических расчетов в Ruby определен модуль Math.

6.1. Numeric

Добавленные модули: Comparable
Абстрактный класс для работы с числами.
Если в результате выполнения какого-либо выражения интерпретатор возвращает число, то оно автоматически переводится в десятичную систему счисления.

6.1.1. Приведение типов

Неявное приведение

Подклассы чисел за внешней схожестью имеют различную внутреннюю реализацию. Поэтому перед вызовом метода, интерпретатор приводит переданные аргументы к одному классу. Делается это в соответствии с приведенным ниже списком:

1. Если одно из чисел - комплексное, то и другие числа будут преобразованы в комплексные;
2. Если одно из чисел - десятичная дробь, то и другие числа будут преобразованы в десятичные дроби;
3. Если одно из чисел - рациональная дробь, то и другие числа будут преобразованы в рациональные дроби.

Результат будет экземпляром того же класса, к которому приводятся аргументы.

Явное приведение

.coerce(number) *# -> array*

Используется для преобразования двух чисел к одному типу.

1. Если числа принадлежат к разным классам, то они преобразуются в десятичные дроби с помощью метода `.Float`;
2. Текст преобразуется, если он содержит только цифры (поддерживаются двоичная и шестнадцатеричная системы счисления).

```
1.coerce 2.1 # -> [2.1, 1.0]
1.coerce "2.1" # -> [2.1, 1.0]
1.coerce "0xAF" # -> [175.0, 1.0]
1.coerce "q123" # -> error!
```

.i *# -> complex*

Преобразование числа в комплексное. Метод удален из класса Complex.

```
1.i # -> (0+1i)
```

.to_c *# -> complex*

Преобразование числа в комплексное.

```
1.to_c # -> (1+0i)
```

.to_int *# -> integer*

Преобразование числа в целое с помощью метода `number.to_i`.

```
2.1.to_int # -> 2
```

6.1.2. Операторы

.%(number)

Синонимы: `modulo`

Используется для вычисления остатка от деления.

.+(number) Унарный плюс.

.-(number) Унарный минус.

.<=>(number) Сравнение.

6.1.3. Округление

.ceil *# -> integer*

Используется для нахождения наименьшего целого числа, которое будет больше или равно текущего (округление в большую сторону).

```
2.1.ceil # -> 3
```

.floor *# -> integer*

Используется для нахождения наибольшего целого числа, которое будет больше или равно текущего (округление в меньшую сторону).

```
2.1.floor # -> 2
```

.round(precise = 0) *# -> number*

Используется для округления с заданной точностью. Точность определяет разряд, до которого будет выполнено округление.

```
2.11355.round 4 # -> 2.1136
2.round 4 # -> 2.0
```

.truncate *# -> integer*

Целая часть числа.

```
2.1.truncate # -> 2
```

6.1.4. Математические функции

.abs2 *# -> number*

Квадрат числа.

```
-2.1.abs2 # -> 4.41
```

.numerator *# -> integer*

Числитель рациональной дроби, полученной с помощью метода `number.to_r`.

```
2.1.numerator # -> 4728779608739021
```

.denominator *# -> integer*

Знаменатель рациональной дроби, полученной с помощью метода `number.to_r`.

```
2.1.denominator # -> 2251799813685248
```

.divmod(number) *# -> array*

Частное и остаток от деления.

```
1.divmod 3 # -> [0, 1]
```

.div(number) *# -> integer*

Округленное частное (округление выполняется с помощью метода `number.to_i`).

```
1.div 3 # -> 0
```

.fdiv(number) *# -> float*

Частное в виде десятичной дроби.

```
1.fdiv 3 # -> 0.3333333333333333
```

.quo(number) # -> quotient

Частное двух чисел. Для двух целых чисел результатом будет рациональная дробь.

```
1.quo 3 # -> (1/3)
```

.remainder(number) # -> integer

Остаток от деления, вычисляемый как

```
self - number * ( self / number ).truncate.
1.remainder 3 # -> 1
```

.abs # -> number

Модуль числа.

```
-2.1.abs # -> 2.1
```

.arg # -> number

Синонимы: `angle, phase`

Угловое значение для полярной системы координат.

Для действительных чисел возвращает ноль, если число не отрицательно. В другом случае возвращается ссылка на константу `Math::PI`.

```
1.arg # -> 0
-1.arg # -> 3.141592653589793
```

.polar # -> array

Число в полярной системе координат (`[ self.abs, self.arg ]`).

```
1.polar # -> [1, 0]
```

.real # -> number

Вещественная часть числа.

```
1.real # -> 1
```

.imag # -> 0

Синонимы: `imaginary`

Мнимая часть числа.

.rect # -> array

Число в прямоугольной системе координат (`[ self, 0 ]`).

```
1.rect # -> [ 1, 0 ]
```

.conj # -> self

Синонимы: `conjugate`

Сопряженное число (используется для комплексных чисел).

6.1.5. Предикаты

.integer? *# -> bool*

Проверка относится ли число к целым.

```
1.integer? # -> true
2.1.integer? # -> false
```

.real? *# -> bool*

Проверка относится ли число к действительным (отрицательный результат возвращается только для комплексных чисел).

```
1.real? # -> true
```

.nonzero? *# -> bool*

Сравнение с нулем. Возвращает число, если оно не равно нулю.

```
1.nonzero? # -> 1
0.0.nonzero? # -> nil
```

.zero? *# -> bool*

Сравнение с нулем.

```
1.zero? # -> false
0.0.zero? # -> true
```

6.1.6. Итераторы

.step(limit, step = 1) { |number| } *# -> self*

Перебор чисел.

Если одно из чисел не относится к целым, то все числа преобразуются в десятичные дроби. При этом число итераций соответствует
`n + n * Flt::EPSILON`, где `n == limit - self / step`.

6.1.7. Остальное

.singleton_method_added(*object)

Попытка определить собственный метод числа считается исключением.

6.2. Целые числа (Integer)

Абстрактный класс для работы с целыми числами. Производные классы Fixnum и Bignum отличаются только внутренней реализацией.

6.2.1. Приведение типов

.to_i *# -> self*

.to_r *# -> rational*

Синонимы: `rationalize`

Преобразование в рациональную дробь.

```
0.to_r # -> (0/1)
```

.to_f *# -> float [Fixnum и Bignum]*

Преобразование в десятичную дробь.

```
1.to_f # -> 1.0
```

.to_s(numeral_system = 10) *# -> string [Fixnum и Bignum]*

Преобразование в текст, используя указанную систему счисления (от 2 до 36).

```
16.to_s 16 # -> "10"
0xF.to_s 16 # -> "f"
0x16.to_s 16 # -> "16"
```

6.2.2. Операторы: Fixnum и Bignum

.*(integer) *# -> result* Произведение.

./(integer) *# -> result* Деление.

.(integer)** *# -> result* Возведение в степень.

.+(integer) *# -> result* Сумма.

.-(integer) *# -> result* Разность.

.~(integer) *# -> result* Побитовое отрицание.

.[](index) *# -> 0 или 1*

Используется для получения указанного бита из двоичного представления числа.

.«(integer) *# -> result* Побитовый сдвиг влево.

.»(integer) *# -> result* Побитовый сдвиг вправо.

.&(integer) *# -> result* Побитовое И.

.|(integer) *# -> result* Побитовое ИЛИ.

.^(integer) *# -> result* Побитовое исключающее ИЛИ.

6.2.3. Арифметические операции

.next *# -> integer*

Синонимы: `succ`

Увеличение числа на единицу.

```
1.next # -> 2
```

.pred *# -> integer*

Уменьшение числа на единицу.

```
1.pred # -> 0
```

.gcd(integer) *# -> gcd*

Используется для вычисления наибольшего общего делителя двух целых чисел. Если одно из них равно нулю, то возвращается результат вызова метода `integer.abs` для другого.

```
2.gcd 3 # -> 1
```

.lcm(integer) *# -> lcm*

Используется для вычисления наименьшего общего кратного двух целых чисел. Если одно из них равно нулю, то возвращается ноль.

```
2.lcm 3 # -> 6
```

.gcdlcm(integer) *# -> array*

Используется для вычисления `[ self.gcd(integer), self.lcm(integer) ]`

```
2.gcdlcm 3 # -> [ 1, 6 ]
```

6.2.4. Предикаты

.even? *# -> bool*

Проверка относится ли число к четным.

```
0.even? # -> true
1.even? # -> false
2.even? # -> true
```

.odd? *# -> bool*

Проверка относится ли число к нечетным.

```
0.odd? # -> false
1.odd? # -> true
2.odd? # -> false
```

6.2.5. Итераторы

.upto(limit) { |integer| } *# -> self*

Перебор чисел (включительно) с шагом +1.

.downto(limit) { |integer| } *# -> self*

Перебор чисел (включительно) с шагом -1.

.times { |integer| } *# -> self*

Перебор чисел из диапазона `0...self`.

6.2.6. Остальное

.chr(encode = "binary") *# -> string*

Используется для получения символа с переданной кодовой позицией. Отсутствие символа считается исключением `RangeError`.

```
42.chr # -> "*"
```

.hash *# -> integer [Fixnum и Bignum]*

Цифровой код объекта.

```
1.hash # -> -861462684
```

.size *# -> integer [Fixnum и Bignum]*

Количество байтов, занимаемых числом.

```
1.size # -> 4
```

6.3. Десятичные дроби (Float)

Десятичные дроби реализованы в Ruby как числа с плавающей точкой.

Константы:

Float::ROUNDS - способ округления чисел по умолчанию;
Float::RADIX - показатель степени для представления порядка числа;
Float::MANT_DIG - количество цифр в мантиссе;
Float::DIG - максимально возможная точность;
Float::MIN_EXP - минимально возможный показатель степени 10;
Float::MAX_EXP - максимально возможный показатель степени;
Float::MIN_10_EXP - минимально возможная экспонента;
Float::MAX_10_EXP - максимально возможная экспонента;
Float::MIN - минимально возможная десятичная дробь;
Float::MAX - максимально возможная десятичная дробь;
Float::EPSILON - минимальное число, при добавлении к которому единицы, в результате не возвращается 1.0;

Float::INFINITY - используется для бесконечности;
Float::NAN - инициализируется в результате выполнения выражения `0.0 / 0.0`.

6.3.1. Приведение типов

.to_f *# -> float*

.to_i *# -> integer*

Целая часть десятичной дроби.

```
2.1.to_i # -> 2
```

.to_r *# -> rational*

Преобразование десятичной дроби в рациональную с максимально возможной точностью.

```
2.1.to_r # -> ( 4728779608739021 / 2251799813685248 )
```

.rationalize(number = Flt::EPSILON) *# -> rational*

Преобразование десятичной дроби в рациональную, так что

```
( self - number.abs ) <= rational and rational <= ( self + number.abs )
2.1.rationalize # -> (21/10)
```

.to_s *# -> string*

Преобразование десятичной дроби в текст. Допускается возвращение `"NaN"`, `"+Infinity"` или `"-Infinity"`.

```
2.1.to_s # -> "2.1"
```

6.3.2. Операторы

.*(number) *# -> result* Произведение.

./(number) *# -> result* Деление.

.(number)** *# -> result* Возведение в степень.

.+(number) *# -> result* Сумма.

.-(number) *# -> result* Разность.

6.3.3. Предикаты

.finite? *# -> bool*

Проверка относится ли десятичная дробь к конечным дробям.

```
2.1.finite? # -> true
```

.infinite? *# -> -1, nil, 1*

Используется для вычисления направления бесконечности десятичной дроби. Если дробь относится к конечным дробям, то, в результате, возвращается nil.

```
2.1.infinite? # -> nil
```

.nan? *# -> bool*

Проверка ссылается ли десятичная дробь на константу `Float::NAN`.

```
2.1.nan? # -> false
```

6.3.4. Остальное

.hash *# -> integer*

Цифровой код объекта.

```
2.1.hash # -> 569222191
```

6.4. Рациональные дроби (Rational)

Рациональная дробь - это рациональное число, вида `(a/b)`, где число a называют числителем, а число b - знаменателем. Косая черта обозначает деление двух чисел. Рациональные дроби используются чтобы избежать ошибок приближения при работе с десятичными дробями.

.Rational(nom, denom = 1) *# -> rational [Kernel]*

Используется для создания рациональных дробей вида `(nom/denom)`.

```
Rational 2, 3 # -> (2/3)
Rational "2/3" # -> (2/3)
Rational 8, 3 # -> (8/3)
Rational 2.1, 3 # -> (4728779608739021/6755399441055744)
```

6.4.1. Приведение типов

.to_r *# -> rational*

.inspect *# -> string*

Преобразование в текст.

```
Rational( 2, 3 ).inspect # -> "(2/3)"
```

.to_s *# -> string*

Преобразование тела рациональной дроби в текст.

```
Rational( 2, 3 ).to_s # -> "2/3"
```

.to_f *# -> float*

Преобразование рациональной дроби в десятичную.

```
Rational( 2, 3 ).to_f # -> 0.6666666666666666
```

.to_i *# -> integer*

Используется для получения целой части числа.

```
Rational( 2, 3 ).to_i # -> 0
```

6.4.2. Операторы

.*(number) *# -> result* Произведение.

.(number)** *# -> result* Возведение в степень.

.+(number) *# -> result* Сумма.

.-(number) *# -> result* Разность.

./(number) *# -> result*

Синонимы: `quo`

Деление.

6.4.3. Остальное

.rationalize(number = Flt::EPSILON) *# -> rational*

Преобразование рациональной дроби, так что

```
self - number.abs <= rational && rational <= self + number.abs
```

6.5. Комплексные числа (Complex)

Комплексные числа - это подвид вещественных чисел в виде суммы `(a+b*i)`, где a и b - вещественные числа, а i - мнимая единица.

В классе удалены некоторые базовые методы из Numeric (все методы, относящиеся к округлению чисел, оператор `%`, методы `number.div`, `number.divmod`, `number.remainder`, итератор `number.step`).

Константы:

Complex::I - мнимая единица.

.Complex(real, imag = 0) *# -> complex [Kernel]*

Используется для создания комплексных чисел.

```
Complex 2, 3 # -> (2+3i)
Complex "2/3" # -> ((2/3)+0i)
Complex 8, 3 # -> (8+3i)
Complex 2.1, 3 # -> (2.1+3i)
Complex ?i # -> (0+1i)
```

::polar(magnitude, angle = 0.0) *# -> complex*

Используется для создания комплексного числа в полярной системе координат.

```
Complex.polar 2, 3 # -> (-1.9799849932008908+0.2822400161197344i)
```

::rect(real, imag = nil) *# -> complex*

Синонимы: `rectangular`

Используется для создания комплексного числа в прямоугольной системе координат.

```
Complex.rect 2, 3 # -> (2+3i)
```

6.5.1. Приведение типов

.inspect *# -> string*

Преобразование в текст.

```
Complex( 2, 3 ).inspect # -> "(2+3i)"
```

.to_s *# -> string*

Преобразование тела комплексного числа в текст.

```
Complex( 2, 3 ).to_s # -> "2+3i"
```

.to_f *# -> float*

Преобразование комплексного числа в десятичную дробь, если такое преобразование возможно.

```
Complex( 2, 3 ).to_f # -> error!
```

.to_i *# -> integer*

Преобразование комплексного числа в целое, если такое преобразование возможно.

```
Complex( 2, 3 ).to_i # -> error!
```

.to_r *# -> rational*

Преобразование комплексного числа в рациональную дробь, если такое преобразование возможно.

```
Complex( 2, 3 ).to_r # -> error!
```

.rationalize(number = Flt::EPSILON) *# -> rational*

Преобразование комплексного числа в рациональную дробь, если такое преобразование возможно. Переданный аргумент игнорируется.

```
Complex(3).rationalize true # -> (3/1)
```

6.5.2. Операторы

.*(number) *# -> result*

Произведение.

```
Complex( 2, 3 ) * 2 # -> (4+6i)
```

./(number) *# -> result*

Синонимы: quo

Деление.

```
Complex( 2, 3 ) / 2 # -> ((1/1)+(3/2)*i)
```

.(number)** *# -> result*

Возведение в степень.

```
Complex(2, 3)**2 # -> (-5+12i)
```

.+(number) *# -> result*

Сумма.

```
Complex( 2, 3 ) + 2 # -> (4+3i)
```

.-(number) *# -> result*

Разность.

```
Complex( 2, 3 ) - 2 # -> (0+3i)
```

.- *# -> result*

Унарный минус.

```
-Complex( 2, 3 ) # -> (-2-3i)
```

6.5.3. Математические функции

.abs2 *# -> number*

Квадрат абсолютного значения.

```
Complex( 2, 3 ).abs2 # # -> 13
```

.numerator *# -> complex*

Числитель возможной рациональной дроби.

```
Complex( 2, 3 ).numerator # -> (2+3i)
```

.denominator *# -> complex*

Знаменатель возможной рациональной дроби (наименьшее общее кратное рациональной и мнимой частей).

```
Complex( 2, 3 ).denominator # -> 1
```

.fdiv(number) *# -> complex*

Используется для вычисления частного каждой части в виде десятичной дроби.

```
Complex( 2, 3 ).fdiv 2 # -> (1.0+1.5i)
```

.abs *# -> number*

Синонимы: `magnitude`

Абсолютная часть в полярной системе координат.

```
Complex( 2, 3 ).abs # -> 3.605551275463989
```

.arg *# -> float*

Синонимы: `angle, phase`

Угловое значение в полярной системе координат.

```
Complex( 2, 3 ).arg # -> 0.982793723247329
```

.real *# -> number*

Вещественная часть числа.

```
Complex( 2, 3 ).real # -> 2
```

.imag *# -> number*

Синонимы: `imaginary`

Мнимая часть числа.

```
Complex( 2, 3 ).imag # -> 3
```

.rect *# -> float*

Синонимы: `rectangular`

Вещественная и мнимая части числа в прямоугольной системе координат.

```
Complex( 2, 3 ).rect # -> [2, 3]
```

.conj *# -> complex*

Синонимы: `conjugate`

Сопряженное комплексное число.

```
Complex( 2, 3 ).conj # -> (2-3i)
```

6.6. Math

Модуль содержит определение различных математических функций. Все методы могут быть вызваны либо как методы класса (для модуля Math), либо как частные методы экземпляров (для любого класса, включающего модуль Math).

Переданные аргументы, преобразуются в десятичные дроби. Поэтому в результате вызова метода также возвращается десятичная дробь.

Константы:

Math::PI - число π (пи);
Math::E - число e.

.acos(number) *# -> float* Арккосинус.

.acosh(number) *# -> float* Гиперболический косинус.

.asin(number) *# -> float* Арксинус.

.asinh(number) *# -> float* Гиперболический синус.

.atan(number) *# -> float* Арктангенс.

.atan2(first_number, second_number) *# -> float* Арктангенс.

.atanh(number) *# -> float* Гиперболический тангенс.

.cbrt(number) *# -> float* Кубический корень.

.cos(number) *# -> float* Косинус (в радианах).

.cosh(number) *# -> float* Гиперболический косинус (в радианах).

.erf(number) *# -> float* Функция ошибки.

.erfc(number) *# -> float* Дополнительная функция ошибки.

.exp(number) *# -> float* Возведение e в степень.

.frexp(number) *# -> array*

Индексный массив вида `[ float, integer]`,
где `float * 2**integer == number`.

.gamma(number) *# -> float* Гамма функция из числа.

.hypot(first_number, second_number) *# -> float* Длина гипотенузы.

.ldexp(float, integer) *# -> float*

Аналогично выполнению `float * 2**integer`.

.lgamma(number) *# -> array*

Индексный массив вида
`[ Math.log( Math.gamma(number).abs), Math.gamma(number) < 0 ? -1: 1 ]`.

.log(number, base = Math::E) *# -> float* Логарифм

.log10(number) *# -> float* Десятичный логарифм.

.log2(number) *# -> float* Логарифм по основанию 2.

.sin(number) *# -> float* Синус (в радианах).

.sinh(number) *# -> float* Гиперболический синус (в радианах).

.sqrt(number) *# -> float* Квадратный корень.

.tan(number) *# -> float* Тангенс (в радианах).

.tanh(number) *# -> float* Гиперболический тангенс (в радианах).

Глава 7

Текст

7.1. Текст (String)

Добавленные модули: Comparable

При работе с текстом следует помнить, что интерпретатор на стадии создания объекта не анализирует его значение. Поэтому для любого текста, всегда создается новый объект, даже если объект с таким же значением уже существует.

::new(string = ””) *# -> string*

Используется для создания нового объекта.

```
String.new # -> ""
```

7.1.1. Приведение типов

::try_convert(object) *# -> string*

Преобразование в текст с помощью метода `.to_str`. Если для объекта этот метод не определен, то возвращается nil.

```
String.try_convert [1] # -> nil
```

.to_s *# -> string*

Синонимы: `to_str`

.to_sym *# -> sym*

Преобразование в идентификатор.

```
"abc".to_sym # -> :abc
"123a".to_sym # -> :"123a"
```

.to_i(numeral_system = 10) *# -> integer*

Преобразование в целое число в заданной системе счисления. Обработка продолжается до первого символа, не относящегося к цифрам. Если текст начинается с такого символа или преобразование невозможно, то возвращается 0.

```
"1".to_i # -> 1
"1a".to_i # -> 1
"1x".to_i # -> 1
"1.2".to_i # -> 1
"4/2".to_i # -> 4
"1 + 2".to_i # -> 1
"1   2".to_i # -> 1
"1e2".to_i # -> 1
"1_2".to_i # -> 12
"0b01 ax".to_i # -> 0
"0x01 ax".to_i # -> 0
"1+1i".to_i # -> 1
```

.to_r *# -> rational*

Преобразование в рациональную дробь. Обработка продолжается до первого символа, не относящегося к цифрам. Если текст начинается с такого символа или преобразование невозможно, то возвращается `(0/1)`.

```
"1".to_r # -> 1/1
"1a".to_r # -> 1/1
"1x".to_r # -> 1/1
"1.2".to_r # -> 6/5
"4/2".to_r # -> 2/1
"1 + 2".to_r # -> 1/1
"1   2".to_r # -> 1/1
"1e2".to_r # -> 100/1
"1_2".to_r # -> 12/1
"0b01 ax".to_r # -> 0/1
"0x01 ax".to_r # -> 0/1
"1+1i".to_r # -> 1/1
```

.to_f *# -> float*

Преобразование в десятичную дробь. Обработка продолжается до первого символа, не относящегося к цифрам. Если текст начинается с такого символа или преобразование невозможно, то возвращается `0.0`.

```
"1".to_f # -> 1.0
"1a".to_f # -> 1.0
"1x".to_f # -> 1.0
"1.2".to_f # -> 1.2
"4/2".to_f # -> 4.0
"1 + 2".to_f # -> 1.0
"1   2".to_f # -> 1.0
"1e2".to_f # -> 100.0
"1_2".to_f # -> 12.0
"0b01 ax".to_f # -> 0.0
"0x01 ax".to_f # -> 0.0
"1+1i".to_f # -> 1.0
```

.to_c *# -> complex*

Преобразование в комплексное число. Обработка продолжается до первого символа, не относящегося к цифрам. Если текст начинается с такого символа или преобразование невозможно, то возвращается `(0+0i)`.

```
"1".to_c # -> 1+0i
"1a".to_c # -> 1+0i
"1x".to_c # -> 1+0i
"1.2".to_c # -> 1.2+0i
"4/2".to_c # -> 2/1+0i
"1 + 2".to_c # -> 1+0i
"1   2".to_c # -> 1+0i
"1e2".to_c # -> 100.0 + 0i
"1_2".to_c # -> 12+0i
"0b01 ax".to_c # -> 0+0i
"0x01 ax".to_c # -> 0+0i
"1+1i".to_c # -> 1+1i
```

.hex *# -> integer*

Преобразование в число в шестнадцатеричной системе счисления. Обработка продолжается до первого символа, не относящегося к цифрам. Если текст начинается с такого символа или преобразование невозможно, то возвращается 0.

```
"1".hex # -> 1
"1a".hex # -> 26
"1x".hex# -> 1
"1.2".hex # -> 1
"4/2".hex # -> 4
"1 + 2".hex # -> 1
"1   2".hex # -> 1
"1e2".hex # -> 482
"1_2".hex # -> 18
"0b01 ax".hex # -> 2817
"0x01 ax".hex # -> 1
"1+1i".hex # -> 1
```

.oct *# -> integer*

Преобразование в число в восьмеричной системе счисления. Обработка продолжается до первого символа, не относящегося к цифрам. Если текст начинается с такого символа или преобразование невозможно, то возвращается 0.

```
"1".oct # -> 1
"1a".oct # -> 1
"1x".oct # -> 1
"1.2".oct # -> 1
"4/2".oct # -> 4
"1 + 2".oct # -> 1
"1   2".oct # -> 1
"1e2".oct # -> 1
"1_2".oct # -> 10
"0b01 ax".oct # -> 1
"0x01 ax".oct # -> 1
"1+1i".oct # -> 1
```

7.1.2. Элементы

Любой текст может быть обработан как индексный массив, содержащий отдельные символы в качестве элементов.

В классе String определены операторы `[]` и `[]=`, использующиеся для получения и изменения части текста. Индексация символов начинается с нуля. Если индекс отрицательный, то отсчет символов ведется справа налево, начиная с -1.

string.[*object]

Синонимы: `slice(*object)`

.[index] *# -> string*

Используется для получения символа с заданным индексом. Если индекс выходит за пределы текста, то возвращается nil.

```
"abc"[2] # -> "c"
"abc"[4] # -> nil
```

.[start, length] *# -> string*

Используется для получения фрагмента текста.

- Когда количество символов выходит за пределы текста, возвращается текст до последнего символа;
- Когда количество символов равно нулю, возвращается пустой текст ("");
- Когда количество символов отрицательно, то возвращается nil;
- Когда индекс выходит за пределы текста, то возвращается пустой текст ("");

```
"abc"[ 2, 1 ] # -> "c"
"abc"[ 2, 2 ] # -> "c"
"abc"[ 2, 0 ] # -> ""
"abc"[ 2, -1 ] # -> nil
"abc"[ 3, 1 ] # -> ""
```

.[range] *# -> string*

Используется для получения фрагмента текста между заданными позициями.

- Когда конечная граница выходит за пределы текста, возвращается текст до последнего символа;
- Когда конечная граница меньше, чем начальная, возвращается пустой текст ("");
- Когда начальная граница выходит за пределы текста, возвращается nil.

```
"abc"[ 1..3 ] # -> "bc"
"abc"[ 1...3 ] # -> "bc"
"abc"[ 1...5 ] # -> "bc"
"abc"[ 1...0 ] # -> ""
"abc"[ 5...9 ] # -> nil
```

.[template] *# -> string*

Используется для получения фрагмента текста, совпадающего с образцом. Если совпадений не найдено, то возвращается nil.

```
"abc"[ /[b-z]+/ ] # -> "bc"
"abc"[ /b-z+/ ] # -> nil
```

.[reg, group] *# -> string*

Используется для получения фрагмента текста, совпадающего с заданной группой. Если совпадений не найдено, то возвращается nil.

```
"abc"[ /(b)c/, 1 ] # -> "b"
"abc"[ /(b)c/, 3 ] # -> nil
```

string.[*object]=

Метод изменяет объект, для которого был вызван. В результате вызова возвращается заменяемый фрагмент.

.[index]=(string) *# -> string*

Используется для изменения символа с указанным индексом. Выход за пределы текста считается исключением.

```
"abc"[2] = "d" # -> "d"
"abc"[4] = "d"  # -> error!
```

.[start, length]=(string) *# -> string*

Используется для изменения фрагмента текста.

- Выход за пределы текста считается исключением;
- Когда количество символов выходит за пределы текста, заменяются символы до конца текста;
- Когда количество символов равно нулю, выполняется вставка текста;
- Отрицательная длина фрагмента считается исключением.

```
"abc"[ 4, 1 ] = "d" # -> error!

"abc"[ 3, 1 ] = "d" # -> "d"
string # -> "abdc"

"abc"[ 2, 1 ] = "d" # -> "d"
string # -> "abd"

"abc"[ 2, 2 ] = "d" # -> "d"
string # -> "abd"

"abc"[ 2, 0 ] = "d" # -> "d"
string # -> "abdc"

"abc"[ 2, -1 ] = "d" # -> error!
```

.[range]=(string) *# -> string*

Используется для изменения фрагмента текста между заданными позициями.

- Когда конечная граница выходит за пределы текста, заменяются символы до конца текста;
- Когда конечная граница меньше, чем начальная, текст вставляется перед символом с индексом, заданным начальной границей диапазона;
- Выход начальной границы за пределы текста считается исключением.

```
"abc"[ 1...2 ] = "d" # -> "d"
string # -> "adc"

"abc"[1...5] = "d" # -> "d"
string # -> "ad"

"abc"[ 1...0 ] = "d" # -> "d"
string # -> "adbc"

"abc"[ 5...9 ] = "d" # -> error!
```

.[template]=(string) *# -> string*

Используется для изменения фрагмента текста, совпадающего с образцом. Отсутствие совпадений считается исключением.

```
"abc"[ /[b-z]+/ ] = "d" # -> "d"
string # -> "ad"
"abc"[ /b-z+/ ] = "d" # -> error!
```

.[reg, group]=(string) *# -> string*

Используется для изменения фрагмента текста, совпадающего с переданной группой. Отсутствие совпадений считается исключением.

```
"abc"[ /(b)c/, 1 ] = "d" # -> "d"
a # -> "adc"
"abc"[ /(b)c/, 3 ] = "d" # -> error!
```

Остальное

.binslice(start, length = nil) *# -> string*

`(range) # -> string`

Используется для получения фрагмента текста. Вместо индекса первого символа передается порядковый номер байта.

.length *# -> integer* Синонимы: `size`

Количество символов в тексте.

```
"abc".length # -> 3
```

.bytesize *# -> integer*

Количество байтов, занимаемых текстом.

```
"abc".bytesize # -> 3
```

.getbyte(index) *# -> integer*

Байт с переданным индексом. Если индекс байта выходит за пределы текста, то возвращается nil.

```
"abc".getbyte 0 # -> 97
```

.setbyte(index, byte) *# -> integer*

Используется для изменения байта с переданным порядковым номером. Выход за пределы текста считается исключением.

```
"abc".setbyte 0, 120 # -> 120
string # -> "xbc"
```

7.1.3. Операторы

.%(object) *# -> string* Форматирование.

.*(integer) *# -> string* Копирование.

.+(string) *# -> text* Объединение.

.«(string) *# -> text*

Синонимы: `concat`

Добавление.

.<=>(object) Сравнение.

.=~(template) *# -> integer*

Используется для поиска совпадений с образцом. Возвращается индекс символа, с которого совпадение начинается. Если совпадений не найдено, то возвращается nil.

Если переданный объект не относится к регулярным выражениям, то интерпретатор выполняет `object =~ string` и возвращает результат выполнения.

7.1.4. Изменение текста

Изменение регистра

.capitalize *# -> string*

Используется для изменения первого символа на прописной, а всех остальных на - строчные. Обрабатываются только ASCII символы.

```
"aBc".capitalize # -> "Abc"
```

.capitalize! *# -> self*

Версия предыдущего метода, изменяющая значение объекта.

.upcase *# -> string*

Используется для изменения всех символов на прописные. Обрабатываются только ASCII символы.

```
"aBc".upcase # -> "ABC"
```

.upcase! *# -> self*

Версия предыдущего метода, изменяющая значение объекта.

.downcase *# -> string*

Используется для изменения всех символов на строчные. Обрабатываются только ASCII символы.

```
"aBc".downcase # -> "abc"
```

.downcase! *# -> self*

Версия предыдущего метода, изменяющая значение объекта.

.swapcase *# -> string*

Используется для изменения регистра всех символов на противоположный. Обрабатываются только ASCII символы.

```
"aBc".swapcase # -> "AbC"
```

.swapcase! *# -> self*

Версия предыдущего метода, изменяющая значение объекта.

Удаление символов

.clear *# -> self*

Используется для удаления всех символов. Изменяет значение объекта.

```
"abc".clear # -> ""
```

.slice!(*object) *# -> string*

Используется для удаления символов. Принимает те же аргументы, что и оператор `[]=`. Возвращается удаленная часть текста.

```
"abc".slice! 2 # -> "c"
string # -> "ab"
```

.chomp(last = $/) *# -> string*

Используется для удаления последнего символа (по умолчанию - символ перевода строки). `"abc".chomp ?c # -> "ab"`

.chomp!(string = $/) *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был удален, то возвращается nil.

.chop *# -> string*

Используется для удаления последнего символа.

```
"abc".chop # -> "ab"
```

.chop! *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был удален, то возвращается nil.

.strip *# -> string*

Используется для удаления всех пробельных символов (пробел, отступ, перевод строки) из начала и конца текста.

```
" abc ".strip # -> "abc"
```

.strip! *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был удален, то возвращается nil.

.lstrip *# -> string*

Используется для удаления всех пробельных символов (пробел, отступ, перевод строки) из начала текста.

```
" abc ".lstrip # -> "abc "
```

.lstrip! *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был удален, то возвращается nil.

.rstrip *# -> string*

Используется для удаления всех пробельных символов (пробел, отступ, перевод строки) из конца текста.

```
" abc ".rstrip # -> " abc"
```

.rstrip! *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был удален, то возвращается nil.

Добавление символов

.insert(index, string) *# -> self*

Используется для вставки текста на переданную позицию.

```
"abc".insert 2, ?d # -> "abdc"
string # -> "abdc"
```

.prepend(string) *# -> self*

Используется для добавления текста в начало. Изменяет значение объекта.

```
"Ruby".prepend "Pure " # -> "Pure Ruby"
```

.center(length, string = ”~”) *# -> text*

Используется для добавления в начало и конец текста недостающее количество символов (до length). Если ни один символ не был добавлен, то возвращается ссылка на объект, для которого метод был вызван.

```
"abc".center 6, ?! # -> "!abc!!"
```

.ljust(length, string = "~") *# -> text*

Используется для добавления в конец текста недостающее количество символов (до length). Если ни один символ не был добавлен, то возвращается ссылка на объект, для которого метод был вызван.

```
"abc".ljust 6, ?! # -> "abc!!!"
```

.rjust(length, string = "~") *# -> text*

Используется для добавления в начало текста недостающее количество символов (до length). Если ни один символ не был добавлен, то возвращается ссылка на объект, для которого метод был вызван.

```
"abc".rjust 6, ?! # -> "!!!abc"
```

Экранирование символов

.dump *# -> string*

Используется для экранирования спецсимволов. Сам текст при этом экранируется двойными кавычками. Символы, не относящиеся к ASCII кодировке заменяются на их кодовые позиции.

```
"3\\n/2".dump # -> "\"3\\\\n/2\""
"3\\n/2л".dump # -> "\"3\\\\n/2\\u{43b}\""
```

.inspect *# -> string*

Используется для экранирование спецсимволов. Сам текст при этом экранируется двойными кавычками.

```
"3\verb!\n!/2".inspect # -> "\"3\\verb!\\n!/2\""
"3\verb!\n!/2л".inspect # -> "\"3\\verb!\\n!/2л\""
```

Остальное

.next *# -> string*

Синонимы: `succ`

Используется для увеличения кодовой позиции последнего символа на единицу. При этом возможна цепная реакция.

```
"xyz".next # -> "xza"
```

.next! *# -> self*

Синонимы: `succ!`

Версия предыдущего метода, изменяющая значение объекта.

.reverse *# -> string*

Используется для перестановки символов в обратном порядке.

```
"abc".reverse # -> "cba"
```

.reverse! *# -> self*

Версия предыдущего метода, изменяющая значение объекта.

.replace(string) *# -> self*

Синонимы: `initialize_copy`

Используется для изменения значения объекта.

```
"abc".replace ?? # -> "?"
```

.unpack(string) *# -> array*

Используется для распаковки двоичного текста на основе переданной форматной строки.

```
"\xFF\xFE\xFD".unpack "C*" # -> [ 255, 254, 253 ]
```

7.1.5. Поиск совпадений

Поиск

.count(*template) *# -> integer*

Используется для получения количества найденных символов. Для образца позволяется использовать спецсимволы ^ (отрицание) и - (диапазон).

Когда методу передается несколько объектов, выполняется пересечения множеств.

```
"abc".count "a-z", "^c" # -> 2
```

.index(template, start = 0) *# -> integer*

Используется для получения индекса символа, с которого начинается совпадение.

Если совпадений не найдено, то возвращается nil.

```
"abbc".index /b/ # -> 1
```

.rindex(template, start = 0) *# -> integer*

Версия предыдущего метода для поиска справа налево, вплоть до символа с переданным индексом (start).

Если совпадений не найдено, то возвращается nil.

```
"abbc".rindex /b/ # -> 2
```

.match(template, start = 0) *# -> match*

```
( template, start = 0 ) { |match| } # -> object
```

Используется для сохранения информации о поиске совпадений. Если совпадений не найдено, то возвращается nil.

.partition(template) *# -> array*

Используется для получения массива из фрагментов текста: до совпадения, совпадающего с образцом, и после совпадения.

Когда совпадений не найдено, в качестве первого элемента возвращается весь текст, а вместо остальных элементов - пустой текст (””).

```
"abbc".partition /b/ # -> ["a", "b", "bc"]
```

.rpartition(template) *# -> array*

Версия предыдущего метода для поиска справа налево.

```
"abbc".rpartition /b/ # -> ["ab", "b", "c"]
```

.split(sep = $;, size = nil) *# -> array*

Используется для разделения текста на фрагменты на основе переданного разделителя (по умолчанию пробел). Несколько пробельных символов подряд игнорируются.

Когда методу передается пустое регулярное выражение, текст делится на фрагменты посимвольно.

```
"a  b  c".split # -> [ "a", "b", "c" ]
"a  b  c".split // # -> [ "a", " ", " ", "b", " ", " ", "c" ]
"a  b  c".split //, 2 # -> [ "a", "  b  c" ]
```

Удаление совпадений

.delete(*template) *# -> string*

Используется для удаления всех найденных совпадений. Для образца позволяется использовать спецсимволы ^ (отрицание) и - (диапазон).

Когда методу передается несколько объектов, выполняется пересечение множеств.

```
"abc".delete "a-z", "^A-Z" # -> ""
```

.delete!(*template) *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был удален, то возвращается nil.

.squeeze(*template) *# -> string*

Используется для удаления повторяющихся символов. Для образца позволяется использовать спецсимволы ^ (отрицание) и - (диапазон).

Когда методу передается несколько объектов, выполняется пересечение множеств.

```
"aabbcc".squeeze "a-z", "^A-Z" # -> "abc"
```

.squeeze!(*template) *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был удален, то возвращается nil.

Замена совпадений

.gsub(template, replace) *# -> string*

```
(template) { |match| } # -> string
```

Используется для изменения всех найденных совпадений. Совпадения могут заменяться на результат их итерации, значение соответствующего группе

ключа, переданный текст (который может содержать спецсимволы `'\1'` или `'\K <идентификатор>'` для вставки совпадений с группой)

```
"abcab".gsub /(a)b/, '\1' # -> "aca"
"abcab".gsub /(a)b/, 'ab' => ?y  # -> "ycy"
"abcab".gsub( /(a)b/ ) { |match| match.next } # -> "accac"
```

.gsub!(template, replace) *# -> self*

```
(template) { |match| } # -> self
```

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был изменен, то возвращается nil.

.sub(template, replace) *# -> string*

```
(template) { |match| } # -> string
```

Версия метода для изменения только первого совпадения.

```
"abcab".sub /(a)b/, '\1'  # -> "acab"
"abcab".sub /(a)b/, 'ab' => ?y # -> "ycab"
"abcab".sub( /(a)b/ ) { |match| match.next } # -> "accab"
```

.sub!(template, replace) *# -> self*

```
(template) { |match| } # -> self
```

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был изменен, то возвращается nil.

.tr(template, replace) *# -> string*

Используется для изменения всех найденных символов. Для образца позволяется использовать спецсимволы ^ (отрицание) и - (диапазон), а в заменяющем фрагменте только -.

```
"abc".tr "^x-z", "X-Z" # -> "ZZZ"
```

.tr!(template, replace) *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был изменен, то возвращается nil.

.tr_s(template, replace) *# -> string*

Версия метода, удаляющая повторяющиеся символы.

```
"aabbcc".tr_s "^x-z", "X-Z" # -> "Z"
```

.tr_s!(template, replace) *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один символ не был изменен, то возвращается nil.

7.1.6. Предикаты

.empty? *# -> bool*

Проверка является ли текст пустым ("").

```
"abc".empty? # -> false
```

.ascii_only? *# -> bool*

Проверка содержит ли текст только ASCII символы.

```
"Нелло".ascii_only? # -> false
```

.include?(template) *# -> bool*

Проверка в тексте наличия совпадений.

```
"abc".include? "ab" # -> true
```

.end_with?(*template) *# -> bool*

Проверка наличия суффикса. Если методу передается несколько объектов, то выполняется пересечение множеств.

```
"abc".end_with? "a", "c" # -> true
```

.start_with?(*template) *# -> bool*

Проверка наличия приставки. Если методу передается несколько объектов, то выполняется пересечение множеств.

```
"abc".start_with? "a", "c" # -> true
```

7.1.7. Итераторы

.each_byte { |byte| } *# -> self*

Синонимы: `bytes`

Перебор байтов.

Во второй версии Ruby метод `.bytes` возвращает массив байт.

.each_char { |char| } *# -> self* Синонимы: `chars`

Перебор символов.

Во второй версии Ruby метод `.chars` возвращает массив символов.

.each_line(sep = $/) { |line| } *# -> self* Синонимы: `lines`

Перебор строк. Также принимается произвольный разделитель для строк (по умолчанию - символ перевода строки).

Во второй версии Ruby метод `.lines` возвращает массив строк.

.each_codepoint { |point| } *# -> self* Синонимы: `codepoints`

Перебор кодовых позиций.

Во второй версии Ruby метод `.codepoints` возвращает массив кодовых позиций.

.upto(last, ending = false) { |string| } *# -> self*

Перебор либо элементов диапазона `self..last`, либо элементов диапазона `self...last` (если методу передается логическая величина true).

7.1.8. Кодировка символов

.valid_encoding? *# -> bool*

Проверка корректна ли информация о кодировке текста.

.encoding *# -> encoding*

Используется для сохранения информации о кодировке текста.

```
"абв".encoding # ->  #<Encoding:UTF-8>
```

.force_encoding(encoding) *# -> self*

Используется для принудительного изменения информации о кодировке текста.

.encode(encoding = Encoding.default_internal, options = {}) *# -> string*

```
( encoding, result, options = {} ) # -> string
```

Используется для перекодировки текста. Принимаемые опции описаны в приложении.

.encode!(encoding = Encoding.default_internal, options = {}) *# -> self*

```
( encoding, result, options = {} ) # -> self
```

Версия предыдущего метода, изменяющая значение объекта.

.b *# -> copy_string [Ruby 2.0]*

Метод используется для получения копии текста в кодировке ASCII.

7.1.9. Остальное

.ord *# -> integer*

Первый байт в тексте. Пустой текст считается исключением.

```
"abc".ord # -> 97
```

.crypt(salt) *# -> string*

Используется для кодирования текста.

Принимается "соль" вида `/[\w\d./]{2,2}/`.

```
"abc".crypt "z1" # -> "z1Pgo5xjkEf8U"
```

.sum(salt = 16) *# -> integer*

Контрольная сумма. `(сумма всех байт) % 2**salt - 1`.

```
"abc".sum # -> 294
```

.hash *# -> integer*

Цифровой код объекта.

```
"abc".hash # -> -913021130
```

.casecmp(object)

Сравнение объектов (<=>). Регистр символов не учитывается.

7.2. Регулярные выражения

7.2.1. Regexp

Константы

Regexp::IGNORECASE - регистр символов игнорируется (модификатор i);
Regexp::EXTENDED - пробельные символы и комментарии игнорируются (модификатор x);
Regexp::MULTILINE - многострочный режим (модификатор m);
Regexp::FIXEDENCODING - другая кодировка.

::new(template, object = nil) *# -> regexp*

Синонимы: `compile`

Используется для создания нового регулярного выражения. Вторым аргументом передаются константы класса или произвольные объекты:

- когда значение аргумента истинно, регистр символов будет игнорироваться;
- когда передаются символы `?n` или `?N`, в теле регулярного выражения используется ASCII кодировка.

```
Regexp.new "abc", 2 # -> /abc/x
```

::union(*template или array = nil) *# -> regexp*

Используется для объединения нескольких регулярных выражений (объединение множеств). Без аргументов возвращается `/!?/`.

```
Regexp.union [ ?0, ?1, ?2 ] # -> /0|1|2/
```

Преобразование типов

::try_convert(object) *# -> regexp*

Преобразование в регулярное выражение с помощью метода `.to_regexp`. Если метод для объекта не определен, то возвращается nil.

```
Regexp.try_convert "abc" # -> nil
```

.to_s *# -> string*

Текст, содержащий тело регулярного выражения и его модификаторы.

```
/(a-z)/i.to_s # -> "(?i-mx:(a-z))"
```

.inspect *# -> string*

Текст, содержащий регулярное выражение.

```
/(a-z)/i.inspect # -> "/(a-z)/i"
```

.source *# -> string*

Текст, содержащий тело регулярного выражения с экранированными спецсимволами.

```
/(a-z)/i.source # -> "(a-z)"
```

Операторы

.===(string) *# -> bool* Поиск совпадений.

.=~(string) *# -> integer* Поиск совпадений.

.~(regexp) *# -> integer*

Поиск совпадений с последней прочитанной строкой (`$_`).

Поиск совпадений

::last_match *# -> match*

```
(group) # -> string
```

Информация о последнем поиске совпадений. Если совпадений не найдено, то возвращается nil.

.match(text, start = 0) *# -> match*

```
( text, start = 0 ) { |match| } # -> object
```

Используется для сохранения информации о поиске совпадений. Если совпадений не найдено, то возвращается nil.

Кодировка символов

.encoding *# -> encoding*

Используемая кодировка.

```
/(a-z)/i.encoding # -> #<Encoding:US-ASCII>
```

.fixed_encoding? *# -> bool*

Проверка используется ли любая кодировка, кроме ASCII.

```
/(a-z)/i.fixed_encoding? # -> false
```

Остальное

.casefold? *# -> bool*

Проверка игнорирования регистра символов (модификатор i).

```
/(a-z)/i.casefold? # -> true
```

.named_captures *# -> hash*

Массив идентификаторов групп, ассоциируемых с их позициями.

```
/(?<group>a-z)/i.named_captures # -> { "group" => [1] }
```

.names *# -> array*

Массив идентификаторов групп.

```
/(?<group>a-z)/i.names # -> ["group"]
```

.options *# -> integer*

Сумма чисел используемых модификаторов.

```
/(a-z)/i.options # -> 1
```

.hash *# -> integer*

Цифровой код объекта.

```
/(a-z)/i.hash # -> -145911848
```

::escape(text) *# -> string*

Синонимы: `quote`

Используется для экранирования спецсимволов.

При этом `Regexp.new( Regexp.escape string ) =~ string # -> 0`

```
Regexp.escape "'\*?{}.'" # -> "'\\*\\?\\{\\}\\.'"
```

7.2.2. MatchData

Экземпляры класса содержат полную информацию о найденных совпадениях.

Преобразование типов

.to_s *# -> string*

Полный текст найденного совпадения.

.inspect *# -> string*

Текст, содержащий информацию об объекте.

.to_a *# -> array*

Массив найденных совпадений.

Элементы

.[index] *# -> string*

Используется для получения совпадения с группой (полный текст совпадения - элемент с индексом 0).

.[first, last] *# -> array*

Используется для получения массива фрагментов, совпадающих с переданными группами. Когда границы диапазона отрицательны, отсчет групп ведется справа налево.

.[range] *# -> array*

Используется для получения массива фрагментов, совпадающих с переданными группами. Когда границы диапазона отрицательны, отсчет групп ведется справа налево.

.[name] *# -> string*

Используется для получения фрагмента, совпадающего с группой.

.captures *# -> array*

Используется для получения массива фрагментов, совпадающих с нумерованными группами.

.offset(group) *# -> array*

Используется для получения массива индексов символов, с которых началось и которыми закончилось совпадение фрагмента с группой.

.begin(group) *# -> integer*

Используется для получения индекса символа, с которого началось совпадение фрагмента с группой.

.end(group) *# -> integer*

Используется для получения индекса символа, которым заканчивается совпадение фрагмента с группой.

.pre_match *# -> string*

Фрагмент текста перед найденным совпадением.

.post_match *# -> string*

Фрагмент текста после найденного совпадения.

Остальное

.regexp *# -> regexp*

Регулярное выражение, с которым происходило сравнение.

.string *# -> string*

Неизменяемая копия текста, в котором выполнялся поиск совпадений.

.size *# -> integer*

Синонимы: `length`

Количество всех элементов поиска (включая полный текст совпадения).

.names *# -> array*

Массив, содержащий идентификаторы групп регулярного выражения.

.hash *# -> integer*

Цифровой код объекта.

7.3. Кодировка

7.3.1. Кодировка текста (Encoding)

В классе определены константы для каждой поддерживаемой кодировки. Вместо них также могут использоваться заранее определенные синонимы.

```
Encoding::UTF_8 # -> #<Encoding:UTF-8>
```

Поддерживаемые кодировки

::default_external *# -> encoding*

Внешняя кодировку, используемая по умолчанию.

::default_external=(encoding) *# -> encoding*

Используется для изменения внешней кодировки, используемой по умолчанию.

::default_internal *# -> encoding*

Внутренняя кодировкf, используемая по умолчанию.

::default_internal=(encoding) *# -> encoding*

Используется для изменения внутренней кодировки, используемой по умолчанию. Для удаления кодировки передается nil.

::locale_charmap *# -> string*

Системная кодировка.

::list *# -> array*

Массив всех поддерживаемых кодировок.

::name_list *# -> array*

Массив всех синонимов.

::aliases *# -> hash*

Массив синонимов, ассоциируемых с экземплярами класса.

::find(aliases) *# -> encoding*

Используется для поиска кодировки, взаимосвязанной с переданным синонимом. Отсутствие синонима считается исключением (для внутренней кодировки может возвращаться nil).

Синонимы:

- *”external”* - внешняя кодировка;
  ”internal” - внутренняя кодировка; *”locale”* - локальная кодировка пользователя; *”filesystem”* - кодировка файловой системы.

::compatible?(first_string, second_string) *# -> encoding*

Используется для проверки совместимости кодировок в текстах. Когда они совместимы, возвращается кодировка, поддерживаемая обоими. В другом случае возвращается nil.

```
Encoding.compatible? "асции", "utf-8" # -> #<Encoding:UTF-8>
```

Экземпляры

.inspect *# -> string*

Информация об объекте.

```
Encoding::UTF_8.inspect # -> "#<Encoding:UTF-8>"
```

.name *# -> string*

Синонимы: `to_s`

Информация о кодировке.

```
Encoding::UTF_8.name # -> "UTF-8"
```

.names *# -> array*

Массив всех доступных синонимов.

```
Encoding::UTF_8.names
# -> ["UTF-8", "CP65001", "locale", "external", "filesystem"]
```

.ascii_compatible? *# -> bool*

Проверка совместимости с ASCII.

```
Encoding::UTF_8.ascii_compatible? # -> true
```

.dummy? *# -> ruby*

Проверка относится ли кодировка к фиктивным. Для фиктивных кодировок обработка символов не реализована должным образом. Метод используется для динамичных кодировок.

```
Encoding::UTF_8.dummy? # -> false
```

.replicate(name) *# -> encoding*

Используется для копирования кодировки. Использование уже существующего имени считается исключением.

```
Encoding::UTF_8.replicate "утф8" # -> #<Encoding:утф8>
```

7.3.2. Преобразование кодировок (Encoding::Converter)

Константы:

::INVALID_MASK - некоректные байты считаются исключением;
::INVALID_REPLACE - некорректные байты заменяются;
::UNDEF_MASK - неопределенные символы считаются исключением;
::UNDEF_REPLACE - неопределенные символы заменяются;
::UNDEF_HEX_CHARREF - неопределенные символы заменяются на байты `&xHH`;
::UNIVERSAL_NEWLINE_DECORATOR - замена CR (`\r`) и CRLF (`\r\n`) на LF (`\n`);
::CRLF_NEWLINE_DECORATOR - замена LF (`\n`) на CRLF (`\r\n`);
::CR_NEWLINE_DECORATOR - замена LF (`\n`) на CR (`\r`);
::XML_TEXT_DECORATOR
::XML_ATTR_CONTENT_DECORATOR
::XML_ATTR_QUOTE_DECORATOR
::PARTIAL_INPUT - обработка исходного текста как части другого объекта;
::AFTER_OUTPUT - цикличное преобразование исходного текста.

::new(source_enc, dest_enc, options = nil) *# -> converter*

```
(conv_path) # -> converter
```

Используется для создания преобразователя. Принимаются исходная кодировка и требуемая. Дополнительные опции описаны в приложении.

Массив считается путем преобразования и должен содержать двухэлементные подмассивы для каждого отдельного преобразования. Дополнительными элементами могут влиять на преобразование.

Поиск необходимых кодировок

::asciicompat_encoding(encoding) *# -> encoding*

Используется для получения кодировки, совместимой с переданной и с ASCII. Если это переданная кодировка, то возвращается nil.

```
Encoding::Converter.asciicompat_encoding "utf-8" # -> nil
Encoding::Converter.asciicompat_encoding "utf-16le"
# -> #<Encoding:UTF-8>
```

::search_convpath(source_enc, dest_enc, options = nil) *# -> array*

Путь преобразования.

```
Encoding::Converter.search_convpath "ISO-8859-1", "EUC-JP",
  universal_newline: true
# -> [ [ #<Encoding:ISO-8859-1>, #<Encoding:UTF-8> ],
#   [ #<Encoding:UTF-8>, #<Encoding:EUC-JP> ],
#   "universal_newline" ]
```

Статистика

.inspect *# -> string*

Информация об объекте.

```
Encoding::Converter.new( "ISO-8859-1", "EUC-JP" ).inspect
# -> "#<Encoding::Converter: ISO-8859-1 to EUC-JP>"
```

.convpath *# -> array*

Путь преобразования.

```
Encoding::Converter.new( "ISO-8859-1", "EUC-JP" ).convpath
# -> [ [ #<Encoding:ISO-8859-1>, #<Encoding:UTF-8> ],
#   [ #<Encoding:UTF-8>, #<Encoding:EUC-JP> ] ]
```

.source_encoding *# -> encoding*

Исходная кодировка.

```
Encoding::Converter.new("ISO-8859-1",
  "EUC-JP").source_encoding
# -> #<Encoding:ISO-8859-1>
```

.destination_encoding *# -> encoding*

Требуемая кодировка.

```
Encoding::Converter.new("ISO-8859-1",
  "EUC-JP").destination_encoding
# -> #<Encoding:EUC-JP>
```

.replacement *# -> string*

Текст для замены некорректных байтов или неопределенных символов.

```
Encoding::Converter.new( "ISO-8859-1", "EUC-JP" ).replacement
# -> "?"
```

.replacement=(string) *# -> string*

Используется для изменения текста, заменяющего некорректные байты или неопределенные символы.

.last_error *# -> error*

Последнее полученное исключение.

Преобразование

.convert(text) *# -> string*

Используется для преобразования кодировки. Некорректные байты или неопределенные символы считаются исключением в любом случае.

`.primitive_convert( text, result, start = nil, bytesize = nil, options = nil )# -> symbol`

Используется для преобразования кодировок без изменения значения объекта. Позволяется ограничивать фрагмент, в который сохраняется результат (по умолчанию в конец текста).

Опции:

partial_input: true , исходный текст может быть частью другого объекта;
after_output: true , после получения результата, ожидается новый исходный текст.

Преобразование завершается при выполнении одного из следующих условий (в скобках указан возвращаемый результат):

- Исходный текст содержит некорректные байты (`:invalid_byte_sequence`);
- Неожиданный конец исходного текста.

  Это возможно, если `:partial_input` не задан (`:incomplete_input`);
- Исходный текст содержит неопределенные символы (`:undefined_conversion`);
- Данные выводятся до их записи.

  Это возможно, если ключ `:after_output` не задан (`:after_output`);
- Буфер назначенного объекта полон.

  Это возможно, если bytesize не ссылается nil (`:destination_buffer_full`);
- Исходный текст пуст.

  Это возможно, если `:partial_input` не задан (`:source_buffer_empty`);
- Преобразование завершено (`:finished`).

.primitive_errinfo *# -> array*

Информация о последней ошибке преобразования в виде:

`[ symbol, source_enc, dest_enc, ivalid_byte, undef_char ]`,

где symbol - результат последнего вызова метода `.primitive_convert`.

Другие элементы имеют смысл только для `:invalid_byte_sequence`, `:incomplete_input` или `:undefined_conversion`.

.putback *# -> string*

Фрагмент текста, который будет преобразован при следующем вызове `.primitive_convert`.

.insert_output(string) *# -> nil*

Используется для добавления текста, необходимого к преобразованию. Когда требуемая кодировка сохраняет свое состояние, текст будет преобразован в соответствии с состоянием, которое будет обновлено после преобразования. Этот метод необходимо использовать только если при преобразовании получено исключение.

.finish *# -> string*

Используется для завершения преобразования.

Глава 8

Составные объекты

Добавленные модули: Enumerable

8.1. Array (индексные массивы)

::new(size = 0, object = nil) *# -> array*

```
(array) # -> array
(size) { |index| } # -> array
```

Используется для создания индексного массива заданного размера. Элементы вычисляются с помощью блока или ссылаются на дополнительный аргумент.

```
Array.new 3, ?R # -> ["R", "R", "R"]
Array.new [1, 2] # -> [1, 2]
Array.new(3) { |index| index**2 } # -> [0, 1, 4]
```

::[*object] *# -> array*

Используется для сохранения объектов в массив.

```
Array[1, 2, 3] # -> [1, 2, 3]
```

8.1.1. Приведение типов

.to_a *# -> array*

Синонимы: `to_ary`

::try_convert(object) *# -> array*

Преобразование объекта в индексный массив с помощью вызова метода `object.to_ary`. Если для объекта этот метод не определен, то возвращается nil.

```
Array.try_convert 1 # -> nil
```

.to_s *# -> string*

Синонимы: `inspect`

Преобразование массива в текст.

```
[ 1, 2, 3 ].to_s # -> "[1, 2, 3]"
```

.join(sep = $,) *# -> string*

Используется для объединения элементов (join - ”объединить”, англ.), используя переданный разделитель (по умолчанию nil).

```
[1, 2, 3].join # -> "123"

[
  "#{msg}",
  "Class: <#{e.class}>",
  "Message: <#{e.message.inspect}>",
  "---Backtrace---",
  "#{MiniTest::filter_backtrace(e.backtrace).join("\n")}",
  "---------------",
].join "\n"
```

.pack(format) *# -> string*

Используется для упаковки массива в двоичный текст, с помощью apppack.

```
[-1, -2, -3].pack "C*" # -> "\xFF\xFE\xFD"
```

8.1.2. Элементы

Для доступа к элементам используются операторы `[]` и `[]=`. Индексация элементов начинается с нуля. Если индекс отрицательный, то отсчет элементов ведется справа налево, начиная с -1.

Наиболее частая проблема с массивами - передача индекса, выходящего за пределы массива.

array.[*object]

Синонимы: `slice(*object)`

.[index] *# -> object*

Синонимы: `at`

Используется для получения элемента с заданным индексом. Если индекс выходит за пределы массива, то возвращается nil.

```
[1, 2, 3][2] # -> 3
[1, 2, 3][4] # -> nil
```

.[start, size] *# -> array*

Используется для получения фрагмента массива заданного размера.

- Если количество элементов выходит за пределы массива, то возвращается вся часть массива до последнего элемента;
- Если количество элементов равно нулю, то возвращается ссылка на пустой массив (`[]`);
- Если количество элементов отрицательно, то возвращается nil;
- Если индекс выходит за пределы массива, то возвращается пустой массив.

```
[1, 2, 3][2, 1] # -> [3]
[1, 2, 3][2, 2] # -> [3]
[1, 2, 3][2, 0] # -> [ ]
[1, 2, 3][2, -1] # -> nil
[1, 2, 3][3, 1] # -> [ ]
```

.[range] *# -> object*

Используется для получения указанного фрагмента массива.

- Если конечная граница выходит за пределы массива, то возвращается вся часть массива до последнего элемента;
- Если конечная граница меньше, чем начальная, то возвращается пустой массив;
- Если начальная граница выходит за пределы массива, то возвращается nil.

```
[1, 2, 3][1...3] # -> [2, 3]
[1, 2, 3][1...5] # -> [2, 3]
[1, 2, 3][1...0] # -> [ ]
[1, 2, 3][5...9] # -> nil
```

array.[*object]=

Используется для изменения значения объекта. В результате возвращается измененный элемент или массив элементов.

- Превышение конечной границы массива при водит к его расширению. Промежуточные элементы при этом ссылаются на nil.
- Выход за начальную границу массива считается исключением.

.[index]=(object) *# -> object*

Используется для изменения элемента с заданным индексом.

```
[1, 2, 3][2] = "d" # -> "d"
array # -> [1, 2, "d"]

[1, 2, 3][4] = "d"  # -> "d"
array # -> [1, 2, 3, "d"]
```

.[start, size]=(object) *# -> object*

Используется для изменения фрагмента массива заданного размера.

- Если количество элементов равно нулю, то выполняется вставка элементов.
- Отрицательный размер считается исключением.

```
[1, 2, 3][4, 1] = "d" # -> "d"
array # -> [1, 2, 3, nil, "d"]

[1, 2, 3][3, 1] = "d" # -> "d"
array # -> [1, 2, 3, "d"]

[1, 2, 3][2, 1] = "d" # -> "d"
array # -> [1, 2, "d"]

[1, 2, 3][2, 2] = "d" # -> "d"
array # -> [1, 2, "d"]

[1, 2, 3][2, 0] = "d" # -> "d"
array # -> [1, 2, "d", 3]

[1, 2, 3][2, -1] = "d" # -> error!
```

.[range]=(object) *# -> object*

Используется для изменения указанного фрагмента массива. Если конечная граница меньше, чем начальная, то элементы добавляются перед индексом, заданным начальной границей диапазона.

```
[1, 2, 3][1...2] = "d" # -> "d"
array # ->[1, "d", 3]

[1, 2, 3][1...5] = "d"# -> "d"
array # -> [1, "d"]

[1, 2, 3][1...0] = "d"# -> "d"
array # -> [1, "d", 2, 3]

[1, 2, 3][5...9] = "d"# -> "d"
array # -> [1, 2, 3, nil, nil, "d"]
```

Остальное:

.fetch(index, object) *# -> object2*

```
(index) { |index| } # -> object
```

Аналогично выполнению `array[index]`. Дополнительный аргумент используется при выходе за пределы массива.

```
[ 1, 2, 3 ].fetch 3, 4 # -> 4
```

.values_at(*object) *# -> array*

Аналогично выполнению `array[*object]` для каждого переданного объекта.

```
[ 1, 2, 3 ].values_at 1, 1 # -> [ 2, 2 ]
```

.sample(size = nil) *# -> object*

```
( size = nil, random: a_random ) # -> object [Ruby 2.0]
```

Используется для получения случайного элемента (или массива случайных элементов). Для пустых массивов возвращается nil или пустой массив соответственно. Если переданный размер равен или превышает размеры исходного массива, то в результате элементы просто перестраиваются в случайном порядке.

```
[ 1, 2, 3 ].sample 4 # -> [ 2, 1, 3 ]
```

Необязательный именованный аргумент используется для создания собственного генератора случайных чисел (Ruby 2.0).

.last(size = 1) *# -> object*

Используется для получения последнего элемента или последнего фрагмента.

```
[ 1, 2, 3 ].last 2 # -> [ 2, 3 ]
```

.index(object) *# -> integer*

```
{ |object| } # -> integer
```

Используется для поиска индекса элемента либо равного переданному объекту, либо с положительным результатом выполнения блока.

```
[ 1, 2, 3 ].index { |elem| elem < 3 } # -> 0
```

.rindex(object) *# -> integer*

```
{ |object| } # -> integer
```

Версия предыдущего метода, выполняющая поиск элемента с конца массива.

```
[ 1, 2, 3 ].rindex { |elem| elem < 3 } # -> 1
```

8.1.3. Операторы

.*(integer) *# -> array* Копирование.

.*(sep) *# -> string*

Используется для объединения элементов в текст с использованием переданного разделителя.

```
[1, 2, 3] * ?? # -> "1?2?3"
```

.+(array) *# -> new_array* Объединение элементов.

.-(array) *# -> new_array* Удаление элементов.

.«(object) *# -> self* Добавление элемента. Изменяется значение объекта.

.&(array) *# -> new_array* Пересечение множеств.

.|(array) *# -> array* Объединение множеств.

8.1.4. Изменение массивов

.concat(array) *# -> self*

Метод используется для добавления в массив переданных элементов. Изменяется значение объекта.

```
[ "a", "b" ].concat [ "c", "d" ] # -> [ "a", "b", "c", "d" ]

a = [ 1, 2, 3 ]
a.concat [ 4, 5 ]
a # -> [ 1, 2, 3, 4, 5 ]
```

Работа со стеком

Стек - это структура данных, представляющая из себя список элементов, в котором доступ к следующему элементу может быть получен только после того как был извлечен предыдущий. Стек организуется по принципу LIFO - элемент, который добавлен последним, будет извлекаться самым первым.

В Ruby с любым массивом можно работать как со стеком.

.push(*object) *# -> self*

Метод используется для добавления элементов в конец массива (изменяется значение объекта).

```
a = %w[ a b c ]
a.push 'd', 'e', 'f' # -> [ 'a', 'b', 'c', 'd', 'e', 'f' ]
[ 1, 2, 3 ].push(4).push(5) # -> [ 1, 2, 3, 4, 5 ]
```

.pop(size = 1) *# -> object || array*

Используется для удаления элементов из конца массива (изменяется значение объекта). Когда массив пуст, возвращается nil.

```
a = %w[ a b c d ]
a.pop   # -> 'd'
a.pop 2 # -> [ 'b', 'c' ]
a       # -> ['a']
```

.unshift(*object) *# -> self*

Метод используется для добавления элементов в начало массива (изменяется значение объекта).

```
a = %w[ b c d ]
a.unshift 'a'  # -> [ 'a', 'b', 'c', 'd' ]
a.unshift 1, 2 # -> [ 1, 2, 'a', 'b', 'c', 'd' ]
```

.shift(size = 1) *# -> object || array*

Метод используется для удаления элементов из начала массива (изменяется значение объекта). Когда массив пуст, возвращается nil.

```
args = %w[ -m -q filename ]
args.shift     # -> '-m'
args           # -> [ '-q', 'filename' ]

args = %w[ -m -q filename ]
args.shift 2  # -> [ '-m', '-q' ]
args           # -> [ 'filename' ]
```

Удаление элементов

.clear *# -> self*

Используется для удаления всех элементов. Изменяет значение объекта.

```
[ 1, 2, 3 ].clear # -> [ ]
```

.compact *# -> array*

Используется для удаления элементов, ссылающихся на nil.

```
[ 1, 2, 3 ].compact # -> [ 1, 2, 3 ]
```

.compact! *# -> self*

Версия предыдущего метода, изменяющая значение объекта.

.uniq *# -> array*

Используется для удаления повторяющихся элементов.

```
[ 1, 2, 3, 3, 2, 1 ].uniq # -> [ 1, 2, 3 ]
```

.uniq! *# -> self*

Версия предыдущего метода, изменяющая значение объекта. Если ни один элемент не был удален, то возвращается nil.

.slice!(*object) *# -> delete*

Используется для удаления фрагментов массива `self[*object]`.

```
[1, 2, 3].slice! 1, 1 # -> [2]
```

.delete(object) *# -> removed_object*

```
(object) { }
```

Используется для удаления всех элементов, равных переданному аргументу (изменяется значение объекта). Если ни один элемент не был удален, то возвращается либо nil, либо результат выполнения необязательного блока.

```
[ 1, 2, 3 ].delete(4) { "error!" } # -> "error!"

[ 1, 2, 3, 1 ].delete 1 # -> 1
array # -> [ 2, 3 ]

array = [ 2, 3, 4 ]
[ 1, 2, 3 ].each { |elem| array.delete elem } # -> [ 1, 2, 3 ]
array # -> [4]
```

.delete_at(index) *# -> delete*

Используется для удаления элемента с заданным индексом (изменяется значение объекта). Если ни один элемент не был удален, то возвращается nil.

```
[1, 2, 3].delete_at 1 # -> 2
array # -> [1, 3]
```

.delete_if { |object| } *# -> self*

Синонимы: `reject!`

Используется для удаления всех элементов с положительным значением итерации (изменяется значение объекта). Если ни один элемент не был удален, то возвращается nil.

```
[1, 2, 3].delete_if { |elem| elem < 3 } # -> [3]
```

.select! { |object| } *# -> self*

Используется для сохранения только элементов с положительным значением итерации (изменяется значение объекта). Если ни один элемент не был удален, то возвращается nil. `[1, 2, 3].select! { |elem| elem < 3 } # -> [1, 2]`

Замена элементов

.replace(array) *# -> self*

Синонимы: `initialize_copy`

Используется для замены значения объекта.

```
[1, 2, 3].replace [ ] # -> [ ]
```

.insert(*(index, *object)) *# -> self*

Аналогично выполнению `array[integer] = *object` для каждой пары переданных методу объектов.

```
[ 1, 2, 3 ].insert 1, 2, 3 # -> [ 1, 2, 3, 2, 3 ]
```

.fill(object, start = 0, size = self.size) *# -> self*

```
( start = 0, size = self.size ) { |index| } # -> self
( object, range ) # -> self
(range) { |index| } # -> self
```

Используется для замены всех элементов фрагмента массива заданного размера или диапазона (изменяется значение объекта). Новые объекты либо ссылаются на переданный аргумент, либо вычисляются в результате выполнения блока.

```
[1, 2, 3].fill 1 # -> [1, 1, 1]
```

Остальное

.flatten(deep = nil) *# -> array*

Используется для извлечения элементов из вложенных подмассивов до заданного уровня вложенности. По умолчанию извлекаются все элементы.

```
[ [[1]], [[2]], [[3]] ].flatten # -> [1, 2, 3]
```

.flatten!(deep) *# -> self* Версия предыдущего метода, изменяющая значение объекта.

.rotate(step = 1) *# -> array*

Используется для вращения элементов массива на заданное число позиций - слева направо для положительного аргумента и справа налево для отрицательного.

```
[1, 2, 3].rotate  # -> [2, 3, 1]
[1, 2, 3].rotate -1  # -> [3, 1, 2]
```

.rotate!(step = 1) *# -> self* Версия предыдущего метода, изменяющая значение объекта.

8.1.5. Сортировка массива

.reverse *# -> array*

Перестановка элементов в обратном порядке.

```
[1, 2, 3].reverse # -> [3, 2, 1]
```

.reverse! *# -> self*

Версия предыдущего метода, изменяющая значение объекта.

.shuffle *# -> array*

```
(random: a_random) # -> array [Ruby 2.0]
```

Перестановка элементов в случайном порядке. Необязательный аргумент используется для создания собственного генератора случайных чисел (Ruby 2.0).

```
[ 1, 2, 3 ].shuffle # -> [ 2, 3, 1 ]
[ 1, 2, 3 ].shuffle random: Random.new(1) # -> [ 1, 3, 2 ]
```

.shuffle! *# -> self*

```
(random: a_random) # -> self [Ruby 2.0]
```

Версия предыдущего метода, изменяющая значение объекта.

.sort! *# -> self*

```
{ |object, object2| } # -> self
```

Используется для сортировки элементов (изменяется значение объекта). Элементы сравниваются либо с помощью оператора <=>, либо на основе результатов итераций.

.sort_by! { |object, object2| } *# -> self*

Используется для сортировки элементов (изменяется значение объекта) в восходящем порядке на основе результатов итераций.

8.1.6. Итераторы

.each { |object| } *# -> self* Перебор элементов.

.each_index { |index| } *# -> self* Перебор индексов.

.collect! { |object| } *# -> self*

Синонимы: `map!`

Используется для замены элементов на результат их итерации (изменяется значение объекта). Часто применяется вместе с функциональным стилем программирования.

```
[1, 2, 3].collect! { |elem| elem + 1 } # -> [2, 3, 4]
[1, 2, 3].collect! &:to_s # -> ["1", "2", "3"]
```

.combination(size) { |array| } *# -> self*

Перебор всех возможные фрагментов заданного размера. Различный порядок элементов при этом игнорируется.

- Если аргумент равен нулю, то итерируется `[[]]`;
- Если аргумент больше, чем размер объекта, то итерируется пустой массив.

.repeated_combination(size) { |array| } *# -> self*

Версия предыдущего метода, в которой один и тот же элемент может использоваться несколько раз.

.permutation(size = self.size) { |array| } *# -> self*

Версия метода, учитывающая различный порядок элементов.

.repeated_permutation(size = self.size) { |array| } *# -> self*

Версия предыдущего метода, в которой один и тот же элемент может использоваться несколько раз.

8.1.7. Ассоциативные массивы

.assoc(key) *# -> array*

Используется для поиска вложенного подмассива, первый элемент которого равен переданному аргументу. Если совпадений не найдено, то возвращается nil.

```
[ [:a, 1], [:b, 2], [:a, 3] ].assoc :a # -> [:a, 1]
```

.rassoc(object) *# -> array*

Используется для поиска вложенного подмассива, второй элемент которого равен переданному аргументу. Если совпадений не найдено, то возвращается nil.

```
[ [:a, 1], [:b, 2], [:a, 3] ].rassoc :a # -> nil
```

.transpose *# -> array*

Используется для извлечения из вложенных подмассивов первого и второго элементов. В результате возвращается объект, состоящий из двух вложенных подмассивов. Первый содержит все первые элементы, а второй - оставшиеся элементы.

```
[ [:a, 1], [:b, 1] ].transpose # -> [ [:a, :b], [1, 1] ]
```

8.1.8. Остальное

.product(*array) *# -> array2*

```
(*array) { |part| } # -> array3
```

Используется для получения всех возможных фрагментов размером self.size, созданных из элементов всех используемых массивов. Учитывается разный порядок элементов. Каждый элемент может быть использован в подмассиве только один раз.

- При вызове без аргументов, фрагменты будут состоять из одного элемента;
- Если методу передается пустой массив, то в результате также возвращается пустой массив.

```
[1, 2].product [3] # -> [ [1, 3], [2, 3] ]
[1, 2, 3].product # -> [ [1], [2], [3] ]
[1, 2, 3].product [ ] # -> [ ]
```

.bsearch { |x| } *# -> elem [Ruby 2.0]*

Реализация двоичного поиска (метода деления пополам, дихотомии). Используется для нахождения элемента, который отвечает заданному условию за O(log n), где n - это размер массива. Алгоритм выполняет поиск элемента в отсортированном массиве, используя дробление массива на половины.

Метод реализован как в классическом варианте, так и для использования бисекции. В любом случае массив должен быть монотонным (отсортированным) по отношению к блоку.

- Поиск элемента

  Блок должен возвращать логическую величину для каждого элемента. Массив должен содержать элемент с индексом i, так что:

  – Блок возвращает false для любого элемента, индекс которого меньше чем i.
  – Блок возвращает true для любого элемента, индекс которого больше или равен i.

  В результате возвращается элемент с индексом i. Когда индекс равен размеру массива, то возвращается nil.

```
ary = [0, 4, 7, 10, 12]
ary.bsearch {|x| x >=   4 } # -> 4
ary.bsearch {|x| x >=   6 } # -> 7
ary.bsearch {|x| x >=  -1 } # -> 0
ary.bsearch {|x| x >= 100 } # -> nil
```

- Метод бисекции (метод деления отрезка пополам)

  Простейший численный метод для решения нелинейных уравнений вида `F(x) = 0`.

  Блок должен возвращать число для каждого элемента. Массив должен содержать элементы с индексами i и j (`i <= j`), так что:

  – Блок возвращает положительное число для элементов с индексом `0...i`.
  – Блок возвращает ноль для элементов с индексом `i...j`.
  – Блок возвращает отрицательное число для элементов с индексом `j...size`.

  В результате возвращается любой из элементов с индексом из диапазона `i...j`. Если `i == j`, т.е. нет элементов, отвечающих условию, возвращается nil.

```
ary = [0, 4, 7, 10, 12]
# Поиск элемента из диапазона 4...8
ary.bsearch {|x| 1 - x / 4 } # -> 4 или 7
# Поиск элемента из диапазона 8...10
ary.bsearch {|x| 4 - x / 2 } # -> nil
```

.hash *# -> integer*

Цифровой код объекта.

```
[1, 2, 3].hash # -> -831861323
```

.empty? *# -> bool*

Проверка пуст ли массив.

```
[1, 2, 3].empty? # -> false
```

.size *# -> integer*

Синонимы: `length`

Количество элементов. Результат всегда на единицу больше, чем индекс последнего элемента.

```
[ 1, 2, 3 ].size # -> 3
```

8.2. Hash (ассоциативные массивы)

::new(object = nil) *# -> hash*

```
{ |hash, key| } # -> hash
```

Используется для создания массива с переданным значением по умолчанию.

::[key, object] *# -> hash*

```
[ *[key,object] ] # -> hash
[object] # -> hash
```

Используется для создания массива на основе переданных аргументов.

```
Hash[:Ruby, "languages", :Ivan, "man"]
# -> { Ruby: "languages", Ivan: "man" }

Hash[ [ [:Ruby, "languages"], [:Ivan, "man"] ] ]
# -> { Ruby: "languages", Ivan: "man" }

Hash[Ruby: "languages", Ivan: "man"]
# -> { Ruby: "languages", Ivan: "man" }
```

8.2.1. Приведение типов

.to_hash *# -> hash*

.to_h *# -> self [Ruby 2.0]*

Возвращает объект, для которого был вызван. Когда вызывается для производных классов, получатель преобразуется ассоциативный массив.

::try_convert(object) *# -> hash*

Преобразование объекта в массив, с помощью метода `object.to_hash`. Если для объекта этот метод не определен, то возвращается nil.

```
Hash.try_convert[1] # -> nil
```

.to_s *# -> string*

Синонимы: `inspect`

Преобразование массива в текст. Спецсимволы экранируются.

```
{ a: ?a, "b" => '\n' }.to_s # -> "{:a=>\"a\", \"b\"=>\"\\\\n\"}"
```

.to_a *# -> array*

Преобразование ассоциативного массива в индексный вида `[ *[key, object] ]`. Спецсимволы экранируются.

```
{ a: ?a, "b" => '\n' }.to_a # -> [ [:a, "a"], ["b", "\\n"] ]
```

8.2.2. Элементы

.[key] *# -> object*

Используется для получения значения ключа. Если ключ не найден, то возвращается значение по умолчанию.

```
{ a: ?a, "b" => 1 }[:a] # -> "a"
```

.values_at(*key) *# -> array*

Используется для получения значений нескольких ключей.

```
{ a: ?a, "b" => 1 }.values_at :a, :b, ?b # ->  ["a", nil, 1]
```

.select { |key, object| } *# -> hash*

Используется для получения фрагмента массива, содержащего элементы с положительным результатом итерации.

```
{ a: ?a, "b" => 1 }.select { |key| key == ?b } # -> { "b"=>1 }
```

.key(object) *# -> key*

Используется для получения ключа с переданным значением. Если ключ не найден, то возвращается nil.

```
{ a: ?a, "b" => 1 }.key ?a # -> :a
```

.keys *# -> array*

Массив ключей.

```
{ a: ?a, "b" => 1 }.keys # -> [:a, "b"]
```

.values *# -> array*

Массив значений.

```
{ a: ?a, "b" => 1 }.values # -> [ "a", 1 ]
```

.[key]=(object) *# -> object*

Синонимы: `store`

Используется для изменения содержимого массива.

```
{ a: ?a, "b" => 1 }[:a] = 2 # -> 2
hash # -> { :a => 2, "b" => 1 }
```

.fetch(key, default = nil) *# -> object*

```
(key) { |key| } # -> object
```

Используется для получения значения ключа. Отсутствие ключа считается исключением и может привести либо к остановке выполнения, либо к вычислению блока или дополнительного аргумента.

```
{ a: ?a, "b" => 1 }.fetch :b, ?a # -> "a"
```

8.2.3. Изменение массива

Добавление элементов

.merge(hash) *# -> hash2*

```
(hash) { |key, self_value, arg_value| } # -> hash2
```

Используется для объединения двух массивов. При совпадении ключей предпочтение отдается аргументу или результату выполнения блока.

```
{ a: ?a, "b" => 1 }.merge( { "b" => ?b } ) # -> { a: "a", "b" => "b" }
```

.merge!(hash) *# -> self*

```
(hash) { |key, self_value, arg_value| } # -> self
```

Синонимы: `update`

Версия предыдущего метода, изменяющая значение объекта.

Удаление элементов

.clear *# -> self*

Используется для удаления всех элементов (изменяется значение объекта).

```
{ a: ?a, "b" => 1 }.clear # -> { }
```

.shift *# -> array*

Используется для удаления первого элемента (изменяется значение объекта). В результате возвращается индексный массив вида `[key, value]`.

```
{ a: ?a, "b" => 1 }.shift # -> [ :a, "a" ]
```

.delete(key) *# -> object*

```
(key) { |key| } # -> object
```

Используется для удаление заданного элемента (изменяется значение объекта). В результате возвращается значение ключа. Если ключ не найден, то возвращается значение по умолчанию или результат выполнения необязательного блока.

```
{ a: ?a, "b" => 1 }.delete :a # -> "a"
hash # -> { "b"=>1 }
```

.delete_if { |key, value| } *# -> self*

Используется для удаления всех элементов с положительным результатом итерации (изменяется значение объекта).

```
{ a: ?a, "b" => 1 }.delete_if { |key| key == ?b } # -> { a: "a" }
```

.reject { |key, value| } *# -> hash*

Версия предыдущего метода, не изменяющая значение объекта.

```
{ a: ?a, "b" => 1 }.reject { |key| key == ?b } # -> { a: "a" }
```

.reject! { |key, value| } *# -> self || nil*

Версия предыдущего метода, изменяющая значение объекта. Если ни один объект не был удален, то возвращается nil.

```
{ a: ?a, "b" => 1 }.reject! { |key| key == ?c } # -> nil
```

.keep_if { |key, value| } *# -> self*

Используется для сохранения только элементов с положительным результатом итерации (изменяется значение объекта).

```
{ a: ?a, "b" => 1 }.keep_if { |key| key == ?b } # -> { "b"=>1 }
```

.select! { |key, value| } *# -> self*

Версия предыдущего метода, возвращающая nil, если ни один элемент не был удален.

```
{ a: ?a, "b" => 1 }.select! { |key| key == ?b } # -> { "b"=>1 }
```

Остальное

.replace(hash) *# -> self*

Синонимы: `initialize_copy`

Используется для замены значения объекта.

```
{ a: ?a, "b" => 1 }.replace( { } ) # -> { }
```

.invert *# -> hash*

Используется для смены ключей и их значений местами.

```
{ a: ?a, "b" => 1 }.invert # -> { "a" => :a, 1 => "b" }
```

8.2.4. Предикаты

.has_key?(key)

Синонимы: `include?, key?, member?`

Проверка наличия ключа.

```
{ a: ?a, "b" => 1 }.has_key? :a # -> true
```

.has_value?(object)

Синонимы: `value?`

Проверка наличия значения.

```
{ a: ?a, "b" => 1 }.has_value? :a # -> false
```

.compare_by_identity? *# -> bool*

Проверка сравниваются ли все ключи по их объектам-идентификаторам.

.empty? *# -> bool*

Проверка отсутствия элементов.

```
{ a: ?a, "b" => 1 }.empty? # -> false
```

8.2.5. Итераторы

.each { |key, value| } *# -> self*

Синонимы: `each_pair`

Перебор элементов.

.each_key { |key| } *# -> self* Перебор ключей.

.each_value { |value| } *# -> self* Перебор значений.

8.2.6. Индексные массивы

.assoc(key) *# -> array*

Используется для получения индексного массива, содержащего найденный элемент. Если ключ не найден, то возвращается nil. Сравнение ключей выполняется с помощью оператора ==.

```
{ a: ?a, "b" => 1 }.assoc :a # -> [:a, "a"]
```

.rassoc(object) *# -> array*

Версия предыдущего методы, выполняющая поиск элемента по значению.

```
{ a: ?a, "b" => 1 }.rassoc ?a # -> [:a, "a"]
```

.flatten(deep = 0) *# -> array*

Используется для получения индексного массива, содержащего элементы ассоциативного. Все вложенные индексные массивы будут извлекаться до заданного уровня.

```
{ 1 => "one", 2 => [ [2], ["two"] ], 3 => "three" }.flatten 3
# -> [1, "one", 2, 2, "two", 3, "three"]
```

8.2.7. Остальное

.compare_by_identity *# -> self*

Используется для ограничения доступа к элементам с текстовыми ключами.

```
{ a: ?a, "b" => 1 }.compare_by_identity
# -> { :a => "a", "b" => 1 }

hash[:a] # -> "a"
hash["b"] # -> nil
hash[:b] # -> nil
hash.key 1 # -> "b"
```

.size *# -> integer*

Синонимы: `length`

Количество элементов.

```
{ a: ?a, "b" => 1 }.size # -> 2
```

.default *# -> object* Значение по умолчанию.

.default_proc *# -> proc*

Подпрограмма, выполняющаяся по умолчанию (или nil).

.default=(object) *# -> object*

Используется для изменения значения по умолчанию.

.default_proc=(proc) *# -> proc*

Используется для изменения подпрограммы, выполняющейся по умолчанию. Аргумент nil отменяет выполнение подпрограммы (Ruby 2.0).

.hash *# -> integer*

Цифровой код объекта.

```
{ a: ?a, "b" => 1 }.hash # -> -3034512
```

.rehash *# -> hash* Используется для обновления цифровых кодов ключей.

8.3. Range (диапазоны)

Диапазоны необходимо ограничивать круглыми скобками. В противном случае метод будет вызван только для конечной границы.

::new(first, last, include_last = false) *# -> range*

Используется для создания диапазона с переданными границами. Логическая величина влияет на обработку конечной границы.

```
Range.new 1, 5, 0 # -> 1...5
```

8.3.1. Приведение типов

.inspect *# -> string*

Преобразование диапазона в текст, с помощью вызова метода `.inspect` для каждой границы.

```
(1..3).inspect # -> "1..3"
```

.to_s *# -> string* Преобразование диапазон в текст.

```
(1..3).to_s # -> "1..3"
```

8.3.2. Элементы

.begin *# -> object*

Первый элемент диапазона.

```
(1..3).begin # -> 1
```

.end *# -> object*

Последний элемент диапазона.

```
(1..3).end # -> 3
```

.last(size = nil) *# -> array*

Последний элемент или последний фрагмент.

```
(1..3).last 2 # -> [ 2, 3 ]
```

8.3.3. Операторы

.===(object)

Синонимы: `cover?, member?, include?`

Проверяка входит ли объект в диапазон. `(1..3) === 2 # -> true`

8.3.4. Итераторы

.each { |object| } *# -> self* Перебор элементов с помощью метода `.succ`.

.step(step = 1) { |object| } *# -> self*

Перебор элементов с заданным шагом, либо прибавляя его после каждой итерации, либо используя метод `.succ`.

8.3.5. Остальное

.exclude_end? *# -> bool*

Проверка входит ли конечная граница в диапазон.

```
(1..3).exclude_end? # -> false
```

.bsearch { |x| } *# -> elem [Ruby 2.0]*

Реализация двоичного поиска (метода деления пополам, дихотомии). Используется для нахождения элемента, который отвечает заданному условию за O(log n), где n - это размер диапазона. Алгоритм выполняет поиск элемента, используя дробление диапазона на половины.

Метод реализован как в классическом варианте, так и для использования бисекции.

- Поиск элемента

  Блок должен возвращать логическую величину для каждого элемента. Диапазон должен содержать значение x, так что:

  – Блок возвращает false для любого элемента меньше чем x.
  – Блок возвращает true для любого элемента, больше или равного x.

  В результате возвращается x или nil.

```
ary = [0, 4, 7, 10, 12]
(0...ary.size).bsearch { |i| ary[i] >= 4 } # -> 1
(0...ary.size).bsearch { |i| ary[i] >= 6 } # -> 2
(0...ary.size).bsearch { |i| ary[i] >= 8 } # -> 3
(0...ary.size).bsearch { |i| ary[i] >= 100 } # -> nil

(0.0...Float::INFINITY).bsearch { |x| Math.log(x) >= 0 } # -> 1.0
```

- Метод бисекции (метод деления отрезка пополам)

  Простейший численный метод для решения нелинейных уравнений вида `F(x) = 0`.

  Блок должен возвращать число для каждого элемента. Диапазон должен содержать элементы x и y (`x <= y`), так что:

  – Блок возвращает положительное число для элементов меньше чем x.
  – Блок возвращает ноль для элементов из диапазона `x...v`.
  – Блок возвращает отрицательное число для элементов больше иди равным y.

  В результате возвращается любой из элементов из диапазона `x...y`. Если нет элементов, отвечающих условию, возвращается nil.

```
ary = [0, 100, 100, 100, 200]
(0..4).bsearch { |i| 100 - ary[i] } # -> 1, 2 или 3
(0..4).bsearch { |i| 300 - ary[i] } # -> nil
(0..4).bsearch { |i|  50 - ary[i] } # -> nil
```

.hash *# -> integer*

Цифровой код объекта. `(1..3).hash # -> -337569967`

8.4. Enumerator (перечни)

Перечни - это составные объекты, содержащие группу элементов и о методе, вызов которого привел к их группировке. Индексация элементов начинается с нуля.

Перечень всех элементов может быть получен с помощью итераторов, которым не был передан блок.

::new(object, method, *arg) *# -> enum*

```
{ |enum| } # -> enum
```

Используется для создания перечня. Результат также может быть передан в блок. В теле блока предоставляется возможность добавлять элементы в перечень с помощью выражения `enum << object` (как синоним для yield). Тело блока будет выполняться в момент использования перечня.

```
Enumerator.new( [1, 2, ?R], :delete_at, 2 )
# -> #<Enumerator: [1, 2, "R"]:delete_at(2)>

Enumerator.new { |enum| enum << 3 }
# -> #<Enumerator: <Enumerator::Generator:0x87378e8>:each>
```

.enum_for(method = :each, *arg) *# -> enum*

Синонимы: `to_enum`

Используется для создания перечня элементов текущего составного объекта.
`[1, 2, ?R].enum_for # -> #<Enumerator: [1, 2, "R"]:each>`

Во второй версии Ruby метод принимает блок, с помощью которого может быть вычислен размер перечня (без вычисления его элементов).

```
module Enumerable
  # Дублирование элементов составного объекта.
  def repeat(n)
    raise ArgumentError, "#{n} is negative!" if n < 0
    if block_given?
      each { |*val| n.times { yield *val } }
    else
      # __method__ == :repeat
      enum_for( __method__, n ) { size * n if size }
    end
  end
end

enum = (1..14).repeat(3)
enum.first(4) # -> [1, 1, 1, 2]
enum.size # -> 42
```

8.4.1. Приведение типов

.inspect *# -> string*

Используется для получения информации об объекте.

```
Enumerate.new( [1, 2, ?R], :delete_at, 2 ).inspect
# -> "#<Enumerator: [1, 2, \"R\"]:delete_at(2)>"
```

8.4.2. Элементы перечня

.next *# -> object*

Используется для последовательного доступа к элементам перечня. Достижение конца перечня считается исключением `StopIteration`.

.next_values *# -> array*

Версия предыдущего метода, возвращающая элементы в индексном массиве. Этот метод может быть использован для различия между инструкциями yield и yield nil.

.peek *# -> object*

Используется для получения следующего элемента перечня. Достижение конца перечня считается исключением `StopIteration`.

.peek_values *# -> array*

Версия предыдущего метода, возвращающая элементы в индексном массиве. Этот метод может быть использован для различия между инструкциями yield и yield nil.

.rewind *# -> enum*

Используется для обнуления позиции последнего извлеченного элемента.

8.4.3. Итераторы

.each(start = 0) { |object| } *# -> self* Перебор элементов.

.with_index(start = 0) { |object, index| } *# -> self*

Перебор элементов с их индексами.

.with_object(object) { |object2, object| } *# -> object*

Перебор элементов вместе с дополнительным объектом.

8.4.4. Остальное

.feed(object = nil) *# -> nil*

Используется для изменения результата следующей итерации перечня. При вызове без аргументов, использование инструкции yield возвращает nil.

.size *# -> integer [Ruby 2.0]*

Используется для получения размера перечня без вычисления его элементов. Если вычисление невозможно, то возвращается nil.

```
(1..100).to_a.permutation(4).size # -> 94109400
loop.size # -> Float::INFINITY
(1..100).drop_while.size # -> nil
```

8.4.5. Отложенные вычисления (Enumerator::Lazy)

Класс расширяет понятие перечня (наследует Enemerator).

Во второй версии Ruby добавлена возможность откладывать итерацию элементов составного объекта (возможно бесконечного) до того момента как они потребуются.

К сожалению отложенные вычисления обычно медленнее чем обычные, поэтому их применение должно быть оправдано.

::new(enum, size = nil) { | yielder, *values | } *# -> a_lazy_enum*

Используется для создания объекта. Когда будет вычисляться содержимое перечня, элементы составного объекта будут переданы в блок и смогут быть возвращены в перечень с помощью первого параметра блока.

```
# Принудительное вычисление.
module Enumerable
  def filter_map(&block)
    map(&block).compact
  end
end

# Отложенное вычисление.
class Enumerator::Lazy
  def filter_map
    Lazy.new(self) do | yielder, *values |
      result = yield *values
      yielder << result if result
    end
  end
end

(1..Float::INFINITY).lazy.filter_map{ |i| i*i if i.even? }.first(5)
# -> [ 4, 16, 36, 64, 100 ]
```

.lazy *# -> a_lazy_enum*

Используется для создания перечня, позволяющего выполнять отложенные вычисления.

```
[1,2,3].lazy # -> #<Enumerator::Lazy: [1, 2, 3]>
```

Методы

.force *# -> array*

Синонимы: `to_a`

Используется для принудительного вычисления элементов.

.collect_concat { |object| } *# -> a_lazy_enum*

Синонимы: `flat_map`

Используется для получения нового перечня, содержащего результаты итерации всех элементов текущего.

```
[ 'foo', 'bar' ].lazy.flat_map { |i| i.each_char.lazy }.force
# -> [ 'f', 'o', 'o', 'b', 'a', 'r' ]
```

Результаты итерации будут объединяться в том случае, если они относятся к перечням (отвечают на вызовы методов `.each` и `.force`) или массивам (отвечают на вызов метода `.to_ary`).

```
[ {a:1}, {b:2} ].lazy.flat_map { |i| i }.force
#=> [ {a:1}, {b:2} ]
```

.enum_for(method = :each, *args) *# -> a_lazy_enum*

```
( method = :each, *args ) { |*args| } # -> a_lazy_enum
```

Синонимы: `to_enum`

Аналогично соответствующему методу из модуля Kernel. Используется для того, чтобы методы из модуля Enumerable могли возвращать новый вид перечней, если вызываются для объектов подобного типа.

```
r = 1..Float::INFINITY

# Принудительные вычисления.
r.repeat(2).first(5) # -> [ 1, 1, 2, 2, 3 ]
r.repeat(2).class # -> Enumerator
r.repeat(2).map{ |n| n ** 2 }.first(5) # -> бесконечный цикл!

# Отложенные вычисления.
r.lazy.repeat(2).class # -> Enumerator::Lazy
r.lazy.repeat(2).map{ |n| n ** 2 }.first(5) # -> [ 1, 1, 4, 4, 9 ]
```

Применение

- Итерация бесконечных объектов;
- Работа с большими файлами.

```
lines = File.foreach('a_very_large_file')
            .lazy # чтение только необходимой части.
            .select { |line| line.length < 10 }
            .map(&:chomp)
            .each_slice(3)
            .map { |lines| lines.join(';').downcase }
            .take_while { |line| line.length > 20 }

# Чтение первых трёх строк файла или
# до тех пор пока длина строки не превысит 20 символов.
lines.first(3)

lines.to_a # или...
lines.force # чтение файла
lines.each { |elem| puts elem } # и запись каждой строки.
```

8.5. Enumerable

Модуль содержит методы для работы с составными объектами, реализующими перебор своих элементов с помощью метода `.each`.

Также для некоторых методов может понадобиться определение оператора `<=>`.

Ассоциативные массивы преобразуются в индексные с помощью метода `.to_a`.

8.5.1. Приведение типов

.to_a *# -> array*

Синонимы: `entry`

Преобразование в индексный массив.

8.5.2. Элементы

.first(size = nil) *# -> object*

Используется для получения первого элемента или начального фрагмента. Если методу передается ноль, то возвращается пустой массив.

```
[1, 2, 3].first 2 # -> [1, 2]
```

.take(size) *# -> array*

Используется для получения фрагмента заданного размера. Если методу передается ноль, то возвращается пустой массив.

```
[1, 2, 3].take 2 # -> [1, 2]
```

.drop(size) *# -> array*

Используется для удаления фрагмента заданного размера.

```
[1, 2, 3].drop 2 # -> [3]
```

8.5.3. Сортировка и группировка

.sort *# -> array*

```
{ |first, second| } # -> array
```

Используется для сортировки элементов либо с помощью оператора <=>, либо на основе результатов итерации.

```
{ a: 1, b: 2, c: 3 }.sort # -> [ [:a, 1], [:b, 2], [:c, 3] ]
```

.sort_by { |first, second| } *# -> array*

Используется для сортировки в восходящем порядке на основе результатов итерации.

```
{ a: 1, b: 2, c: 3 }.sort_by { |array| -array[1] }
# -> [ [:c, 3], [:b, 2], [:a, 1] ]
```

.group_by { |object| } *# -> hash*

Используется для группировки элементов на основе результата итерации.

```
[1, 2].group_by { |elem| elem > 4 } # -> { false => [1, 2] }
```

.zip(*object) *# -> array*

```
(*object) { |array| } # -> nil
```

Используется для группировки элементов с одинаковыми индексами. Группы могут передаваться в необязательный блок.

Количество групп равно размеру объекта, для которого метод был вызван. Остальные объекты при необходимости дополняются элементами, ссылающимися на nil.

```
{ a: 1, b: 2 }.zip [1, 2], [1]
# -> [ [ [:a, 1], 1, 1 ], [ [:b, 2], 2, nil ] ]
```

.chunk { |object| } *# -> enum*

```
(buffer) { |object, buffer| } # -> enum
```

Используется для группировки элементов на основе результатов итерации.

Результат выполнения блока для дальнейшего использования может быть сохранен с помощью дополнительного аргумента.

Блок может возвращать специальные объекты:

- *nil* или *:_* - игнорировать элемент;
  :_alone - элемент будет единственным в группе.

.slice_before(object) *# -> enum*

```
{ |object| } # -> enum
(buffer  { |object, buffer| } # -> enum
```

Используется для группировки элементов. Новая группа начинается с элемента равного переданному аргументу (сравнение выполняется с помощью оператора ===), или с положительным результатом итерации.

Первый элемент игнорируется.

В перечне каждая группа объектов сохраняется в виде индексного массива.

8.5.4. Поиск элементов

.count(object = nil) *# -> integer*

```
{ |object| } # -> integer
```

Используется для получения количества элементов, либо равных переданному аргументу, либо с положительным результатом итерации. При вызове без аргументов возвращает количество элементов.

```
[ 1, 2, 3 ].count { |elem| elem < 4 } # -> 3
```

.grep(object) *# -> array*

```
(object) { |object| } # -> array
```

Используется для получения всех элементов, равных переданному аргументу (сравнение выполняется с помощью оператора ===).

При получении блока вместо элементов возвращается результат их итерации.

```
[ 1, 2, 3 ].grep(2) { |elem| elem > 4 } # -> [false]
```

.find_all { |object| } *# -> array*

Синонимы: `select`

Используется для получения всех элементов с положительным результатом итерации.

```
[1, 2, 3].find_all { |elem| elem > 4 } # -> [ ]
```

.reject { |object| } *# -> array*

Используется для удаления всех элементов с положительным результатом итерации.

```
[1, 2, 3].reject { |elem| elem > 4 } # -> [1, 2, 3]
```

.partition { |object| } *# -> array*

Используется для получения фрагментов с различными логическими результатами итерации.

```
[1, 2, 3].partition { |elem| elem > 2 } # -> [ [3], [1, 2] ]
```

.detect(default = nil) { |elem| } *# -> object*

Синонимы: `find`

Используется для поиска первого элемента с положительным результатом итерации. Если искомый элемент не найден, то возвращается либо nil, либо значение по умолчанию.

```
[1, 2, 3].detect { |elem| elem > 4 } # -> nil
```

.find_index(object = nil) *# -> index*

```
{ |object| } # -> index
```

Используется для поиска индекса первого элемента, либо равного переданному аргументу, либо с положительным результатом итерации. Если элемент не найден, то возвращается nil.

```
[1, 2, 3].find_index { |elem| elem > 4 } # -> nil
```

8.5.5. Сравнение элементов

Сравнение выполняется с помощью оператора <=>.

.min *# -> object*

Наименьший элемент.

```
[1, 2, 3].min # -> 1
```

.max *# -> object*

Наибольший элемент.

```
[1, 2, 3].max # -> 3
```

.minmax *# -> array*

Наименьший и наибольший элементы.

```
[1, 2, 3].minmax # -> [1, 3]
```

.min_by { |object| } *# -> object*

Элемент с наименьшим результатом итерации.

```
[1, 2, 3].min_by { |elem| -elem } # -> 3
```

.max_by { |object| } *# -> object*

Элемент с наибольшим результатом итерации.

```
[1, 2, 3].max_by { |elem| -elem } # -> 1
```

.minmax_by { |object| } *# -> array*

Элементы с наименьшим и наибольшим результатами итерации.

```
[1, 2, 3].minmax_by { |elem| -elem } # -> [3, 1]
```

8.5.6. Предикаты

.include?(object)

Синонимы: `member?`

Проверка наличия элемента, равного переданному аргументу (сравнение выполняется с помощью оператора ==).

```
[1, 2, 3].include? 4  # -> false
```

.all? { |object| }

Проверка отсутствия элементов с отрицательным результатом итерации. При отсутствии блока проверяется каждый элемент.

```
[1, 2, 3].all? { |elem| elem < 4 } # -> true
```

.any? { |object| }

Проверка наличия хотя бы одного элемента с положительным результатом итерации. При отсутствии блока проверяется каждый элемент.

```
[1, 2, 3].any? { |elem| elem < 4 } # -> true
```

.one? { |object| }

Проверка наличия только одного элемента с положительным результатом итерации. При отсутствии блока проверяется каждый элемент.

```
[1, 2, 3].one? { |elem| elem < 4 } # -> false
```

.none? { |object| }

Проверка отсутствия элементов с положительным результатом итерации. При отсутствии блока проверяется каждый элемент.

```
[1, 2, 3].none? { |elem| elem < 4 } # -> false
```

8.5.7. Итераторы

.collect { |object| } *# -> array*

Синонимы: `map, collect_concat, flat_map`

Перебор элементов с сохранением результатов итерации.

```
[1, 2, 3].collect { |elem| elem < 4 } # -> [true, true, true]
(1..3).collect(&:next) * ?| # -> "2|3|4"
```

.reverse_each(*arg) { |object| } *# -> self*

Перебор элементов в обратном порядке.

.each_with_index { |object, index| } *# -> self*

Перебор элементов вместе с индексами.

.each_with_object(object) { |elem, object| } *# -> object*

Перебор элементов вместе с переданным объектом.

.each_slice(size) { |array| } *# -> nil*

Перебор фрагментов заданного размера.

.each_cons(size) { |array| } *# -> nil*

Перебор фрагментов заданного размера. После каждой итерации из начала фрагмента будет удаляется элемент, а в конец будет добавлен следующий элемент составного объекта.

.each_entry { |object| } *# -> nil*

Перебор элементов. Несколько объектов, переданных инструкции yield в теле метода `.each`, сохраняются в индексном массиве.

8.5.8. Циклы

.drop_while { |object| } *# -> array*

Выполнение блока для всех элементов кроме первого вплоть до получения отрицательного результата итерации. Возвращаются элементы, итерация которых не выполнялась. Иными словами элементы удаляются пока результат итерации положителен. `[1, 2, 3].drop_while { |elem| elem < 4 } # -> [ ]`

.take_while { |object| } *# -> array*

Выполнение блока для всех элементов кроме первого вплоть до получения отрицательного результата итерации. Возвращаются элементы, итерация которых выполнялась. Иными словами элементы сохраняются пока результат итерации положителен.

```
[1, 2, 3].take_while { |elem| elem < 4 } # -> [1, 2, 3]
```

.cycle(step = nil) { |object| } *# -> nil*

Выполнение блока для всех элементов в бесконечном цикле. Выполнение может быть остановлено с помощью инструкций или ограничения количества циклов. Если методу передано отрицательное число, то вызов метода завершается до выполнения.

8.5.9. Остальное

.lazy *# -> a_lazy_enumerator [Ruby 2.0]*

Используется для создания перечня, позволяющего выполнять отложенные вычисления.

.inject(method) *# -> object*

```
(first, method) # -> object
(first = nil) { |buffer, first| } # -> buffer
```

Синонимы: `reduce`

Используется для объединения элементов либо с помощью указанного метода, либо передавая результат каждой итерации первому параметру блока. Выполнение начинается с первого элемента или с дополнительно переданного аргумента.

Часто используется в функциональном стиле программирования и известно как ”свертка”.

```
[1, 2, 3].inject( 100, :+ ) # -> 106
```

Глава 9

Объекты

9.1. Интроспекция

Интроспекция - это технология, позволяющая определять тип и структуру объектов в процессе выполнения программы.

Интроспекция объектов выполняется с помощью методов экземпляров из классов Object, Class, Module.

В Ruby вся программа может рассматриваться в качестве объекта. Интроспекция программы выполняется с помощью методов классов Class и Module, и методов из модуля Kernel.

9.1.1. Проверка выражений

Для проверки выражений используется инструкция `defined?`, принимающая выражение.

Выражения:

```
Выражение # -> ”expression”;
Глобальная переменная # -> ”global-variable”;
Локальная переменная # -> ”local-variable”;
Переменная класса # -> ”class-variable”;
Переменная экземпляра # -> ”instance-variable”;
Константа # -> ”constant”;
true # -> ”true”;
false # -> ”false”;
nil # -> ”nil”;
self # -> ”self”;
yield # -> ”yield”;
super # -> ”super”;
Выражение присваивания # -> ”assignment”;
Вызов метода # -> ”method”.
```

9.1.2. Тип объекта

.class *# -> class [Object]*

Класс объекта.

```
1.class # -> Fixnum
```

.singleton_class *# -> class [Object]*

Собственный класс объекта.

Для nil, true, false возвращаются NilClass, TrueClass, FalseClass соответственно.

Использование для чисел и идентификаторов считается ошибкой.

```
?1.singleton_class # -> #<Class:#<String:0x921114c>>
```

.nil? *# -> [Object]*

Проверка отсутствия подходящего объекта. Только для nil возвращается true.

.===(object) *# -> [Module]*

Проверка типа объекта.

```
Fixnum === 1 # -> true
Integer === 1 # -> true
Comparable === 1 # -> true
```

Используется в альтернативном синтаксисе предложения case, позволяя проверять сразу несколько объектов.

.is_a?(module) *# -> [Object]*

Синонимы: `kind_of?`

Проверка типа объекта.

```
1.is_a? Fixnum # -> true
1.is_a? Integer # -> true
1.is_a? Comparable # -> true
```

.instance_of?(module) *# -> [Object]*

Проверка типа объекта.

```
1.instance_of? Fixnum # -> true
1.instance_of? Integer # -> false
1.instance_of? Comparable # -> false
```

9.1.3. Иерархия наследования

.superclass *# -> class [Class]*

Базовый класс. Для BasicObject возвращается nil.

```
Fixnum.superclass # -> Integer
```

.name *# -> string [Module]*

Название модуля. Для анонимных модулей возвращается nil.

```
Fixnum.name # -> "Fixnum"
```

.to_s *# -> string [Module]*

Название модуля (по умолчанию). Для анонимных модулей возвращается nil.

```
Fixnum.to_s # -> "Fixnum"
```

.ancestors *# -> array [Module]*

Основная иерархия наследования, включая добавленные модули.

```
Fixnum.ancestors
# -> [Fixnum, Integer, Numeric, Comparable, Object, Kernel, BasicObject]
```

.included_modules *# -> array [Module]*

Список всех добавленных модулей из основной иерархии наследования.

```
Fixnum.included_modules # -> [ Comparable, Kernel ]
```

.include?(module) *# -> [Module]*

Проверка наличия переданного модуля в иерархии наследования.

```
Fixnum.include? Comparable # -> true
```

.<=>(module) *# -> -1. 0, 1 или nil [Module]*

Сравнение положения в иерархии наследования.

- *-1* - если первый модуль добавлен ко второму;
  0 - если два модуля ссылаются на один объект;
  1 - если второй модуль добавлен к первому;
  nil - если два модуля не относятся к одной иерархии наследования.

.<(module) *# -> [Module]*

Используется для вычисления отношений в иерархии наследования. Если два модуля не относятся к одной иерархии наследования, то возвращается nil.

```
Integer < Numeric # -> true
Numeric < Comparable # -> true
Integer < Comparable # -> true
```

.>(module) *# -> [Module]*

Используется для вычисления отношений в иерархии наследования. Если два модуля не относятся к одной иерархии наследования, то возвращается nil.

```
Integer > Numeric # -> false
Numeric > Comparable # -> false
Integer > Comparable # -> false
```

.<=(module) *# -> [Module]*

Используется для вычисления отношений в иерархии наследования. Если два модуля не относятся к одной иерархии наследования, то возвращается nil.

```
Integer <= Integer # -> true
Numeric <= Comparable # -> true
Integer <= Comparable # -> true
```

.>=(module) *# -> [Module]*

Используется для вычисления отношений в иерархии наследования. Если два модуля не относятся к одной иерархии наследования, то возвращается nil.

```
Integer >= Integer # -> true
Numeric => Comparable # -> false
Integer => Comparable # -> false
```

9.1.4. Состояние объекта

.to_s *# -> string [Object]*

Класс и цифровой идентификатор объекта.

.inspect *# -> string [Object]*

Класс и цифровой идентификатор объекта. Используется при выводе объекта. **Во второй версии Ruby** метод `.to_s` больше не вызывается.

.object_id *# -> integer [Object]*

Цифровой идентификатор объекта.

```
1.object_id # -> 3
```

.__id__ *# -> integer [Object]*

Цифровой идентификатор объекта.

```
1.__id__ # -> 3
```

Константы

.constants(inherited = true) *# -> array [Module]*

Идентификаторы всех констант в теле модуля. Логическая величина влияет на наличие унаследованных констант.

.const_defined?(name, inherited = true) *# -> [Module]*

Проверка существования константы. Логическая величина влияет на наличие унаследованных констант.

```
Comparable.const_defined? :Fixnum # -> true
```

.const_get(name, inherited = true) *# -> object [Module]*

Метод используется для получения значения константы. Отсутствие константы считается исключением `NameError`. Логическая величина влияет на наличие унаследованных констант.

Во второй версии Ruby при поиске констант учитывается область видимости.

```
const_get("Foo::Bar") # -> 'Foo Bar!'
```

Переменные класса

.class_variables *# -> array*

Список идентификаторов всех существующих переменных класса.

.class_variable_defined?(name)

Проверка существования переменной класса.

.class_variable_get(name) *# -> object*

Значение переменной класса. Отсутствие переменной считается исключением `NameError`.

Переменные экземпляра

.instance_variables *# -> array*

Идентификаторы всех существующих (инициализированных) переменных экземпляра.

.instance_variable_defined?(name)

Проверка существования переменной экземпляра.

.instance_variable_get(name) *# -> object*

Значение переменной экземпляра. Отсутствие переменной считается исключением `NameError`.

9.1.5. Поведение объекта

.respond_to?(method, include_private = false) *# -> [Object]*

Проверка реакции объекта на вызов метода с переданным идентификатором. Логическая величина влияет на необходимость поиска среди частных методов.

Во второй версии Ruby поиск изначально выполняется только для общих методов. Также во второй версии Ruby метод относится к частным.

Если метод для объекта не определен, то вместо него вызывается метод `.respond_to_missing?`.

.singleton_methods(inherited = true) *# -> array [Object]*

Идентификаторы всех существующих собственных методов.

.methods(inherited = false) *# -> array [Object]*

Идентификаторы всех существующих общих и защищенных методов экземпляров.

.public_methods(inherited = true) *# -> array [Object]*

Идентификаторы всех существующих общих методов экземпляров.

.protected_methods(inherited = true) *# -> array [Object]*

Идентификаторы всех существующих защищенных методов экземпляров.

.private_methods(inherited = true) *# -> array [Object]*

Идентификаторы всех существующих частных методов экземпляров.

.instance_methods(inherited = false) *# -> array [Module]*

Идентификаторы всех существующих общих и защищенных методов экземпляров.

.public_instance_methods(inherited = true) *# -> array [Module]*

Идентификаторы всех существующих общих методов экземпляров.

.protected_instance_methods(inherited = true) *# -> array [Module]*

Идентификаторы всех существующих защищенных методов экземпляров.

.private_instance_methods(inherited = true) *# -> array [Module]*

Идентификаторы всех существующих частных методов экземпляров.

.method_defined?(name) *# -> [Module]*

Проверка существования общего или защищенного метода экземпляров.

```
Fixnum.method_defined? :next -> true
```

.public_method_defined?(name) *# -> [Module]*

Проверка существования общего метода экземпляров.

```
Fixnum.public_method_defined? :next -> true
```

.protected_method_defined?(name) *# -> [Module]*

Проверка существования защищенного метода экземпляров.

```
Fixnum.protected_method_defined? :next -> false
```

.private_method_defined?(name) *# -> [Module]*

Проверка существования частного метода экземпляров.

```
Fixnum.private_method_defined? :next -> false
```

.block_given? *# -> bool [Kernel]*

Синонимы: `iterator?`

Проверка передан ли методу блок.

9.2. Метапрограммирование

Метапрограммирование - это технология, позволяющая создавать программы, в результате работы которых создаются новые программы или программы, изменяющие себя в процессе выполнения.

9.2.1. Выполнение произвольного кода

.binding *# -> binding [PRIVATE: Kernel]*

Используется для сохранения текущего состояния выполнения программы. Этот объект может быть передан методу `kernel.eval`.

.eval(code, file = nil, line = nil) *# -> object [Binding]*

Используется для выполнения произвольного кода в контексте, существовавшим при создании объекта. Дополнительные аргументы обрабатываются при получении исключения.

.eval(code, binding = nil, file = nil, line = nil) *# -> object [PRIVATE: Kernel]*

```
{ } # -> object
```

Используется для выполнения произвольного кода. Дополнительные аргументы обрабатываются при получении исключения.

.module_exec(*arg) { |*arg| } *# -> self [Module]*

Синонимы: `class_exec`

Используется для выполнения блока кода в теле модуля.

.module_eval(code, file = nil, line = nil) *# -> self [Module]*

```
{ } # -> self
```

Синонимы: `class_eval`

Используется для выполнения произвольного кода или блока в теле модуля. Дополнительные аргументы обрабатываются при получении исключения.

.instance_exec(*arg) { |*arg| } *# -> self [Object]*

Используется для выполнения блока в области видимости объекта (псевдопеременная self ссылается на объект).

.instance_eval(code, file = nil, line = nil) *# -> object [Object]*

```
{ } # -> self
```

Используется для выполнения произвольного кода или блока в области видимости объекта (псевдопеременная self ссылается на объект). Дополнительные аргументы обрабатываются при получении исключения.

.tap { |object| } *# -> object [Object]*

Используется для выполнения произвольного блока, принимающего текущий объект и возвращающего его после выполнения.

9.2.2. Вызов метода

.send(name, *arg) *# -> [Object]*

Синонимы: `__send__`

Используется для вызова произвольного метода. Вызов не существующего метода считается исключением.

.public_send(name, *arg) *# -> [Object]*

Используется для вызова произвольного общего метода. Вызов не существующего метода считается исключением. Можно вызвать методы, идентификаторы которых содержат произвольные символы.

9.2.3. Перехват выполнения

Перехват выполнения - это технология, позволяющая изменить стандартное поведение интерпретатора при выполнение тех или иных действий.

Перехват выполнения осуществляется после объявления специальных методов, вызываемых автоматически.

.method_missing(name, *arg)

Выполняется при отсутствии вызываемого метода.

.respond_to_missing?(name, include_private = nil)

Выполняется если при вызове `object.respond_to?` необходимый метод не будет найден.

Во второй версии Ruby метод относится к частным.

.const_missing(name)

Выполняется при использовании несуществующей константы.

.singleton_method_added(name)

Выполняется при объявлении собственного метода объекта.

.singleton_method_removed(name)

Выполняется при удалении собственного метода объекта.

.singleton_method_undefined(name)

Выполняется при запрете вызова собственного метода объекта.

.method_added(name)

Выполняется при объявлении метода.

.method_removed(name)

Выполняется при удалении метода.

.method_undefined(name)

Выполняется при запрете вызова метода.

.inherited(class)

Выполняется при создании производного класса.

.append_features(module)

Метод вызывается для каждого модуля, используемого для добавления методов. По умолчанию константы, методы и переменные класса добавляются только в том случае, если модуль отсутствует в иерархии наследования.

.prepend_features(module)

Метод вызывается для каждого модуля, используемого для переопределения методов. По умолчанию модуль переопределяет константы, методы и переменные класса, только в том случае, если он отсутствует в иерархии наследования.

Метод добавлен во второй версии Ruby.

.extended(module)

Выполняется при использовании модуля для расширения класса.

.included(module)

Выполняется при использовании модуля для добавления методов экземпляров.

.prepended(module)

Метод вызывается при использовании модуля для переопределения методов.

Метод добавлен во второй версии Ruby.

```
module A
  def self.prepended(mod)
    puts "#{self} prepended to #{mod}"
  end
end

module Enumerable
  prepend A
end

# -> выводит "A prepended to Enumerable"
```

9.2.4. Изменение состояния

.const_set(name, object) *# -> object [Module]*

Используется для определения констант.

.remove_const(name) *# -> object [PRIVATE: Module]*

Используется для удаления констант. Встроенные классы и модули не могут быть удалены.

.class_variable_set(sym, obj) *# -> object [Module]*

Используется для определения переменной класса.

.remove_class_variable(name) *# -> object [Module]*

Используется для удаления переменной класса.

.instance_variable_set(name, object) *# -> object [Object]*

Используется для определения переменной экземпляра.

.remove_instance_variable(name) *# -> object [PRIVATE: Object]*

Используется для удаления переменной экземпляра.

9.2.5. Изменение поведения

.define_singleton_method(name, block) *# -> block [Object]*

Используется для определения собственного метода объекта.

.define_method(name) { } *# -> proc [PRIVATE: Object]*

```
(name, method) # -> new_method [Ruby 2.0]
```

Метод используется для определения метода экземпляров. Можно создавать методы, идентификаторы которых содержат произвольные символы.

Переданный блок станет телом нового метода (блок выполянется с помощью `.instance_eval`).

Во второй версии Ruby метод может принимать подпрограммы.

```
class A
  def fred; puts "In Fred"; end

  def create_method(name, &block)
    self.class.send(:define_method, name, &block)
  end

  define_method(:wilma) { puts "Charge it!" }
end

class B < A
  define_method(:barney, instance_method(:fred))
end

a = B.new

a.barney # -> 'In Fred'
a.wilma # -> 'Charge it!'

a.create_method(:betty) { p self }
a.betty # -> '#<B:0x97dccfc>'
```

Также **во второй версии Ruby** метод может вызываться для определения глобальных методов.

```
# 1.9.3
define_method(:hello) { "Hello World!" } # -> NoMethodError!

# 2.0.0
define_method(:hello) { "Hello World!" }
hello # -> "Hello World!"
```

.alias_method(new_name, old_name) *# -> self [PRIVATE: Module]*

Используется для создания синонимов.

.remove_method(name) *# -> self [PRIVATE: Module]*

Используется для удаления метода. Унаследованные методы также не могут быть вызваны.

.undef_method(name) *# -> self [PRIVATE: Module]*

Используется для запрета вызова метода.

9.3. Остальное

9.3.1. Приведение типов

Приведение типа - это создание на основе переданного объекта экземпляра другого класса.

Неявное приведение:

В Ruby существуют соглашения, определяющие процесс приведения типов.

- Для получения объекта в виде, удобном для интерпретатора, используются методы `object.to_i`, `object.to_s`, `object.to_a`, `object.to_f`, `object.to_c`, `object.to_r`, `object.to_h` (ruby 2.0) и т.д.
- Для получения объекта в виде, удобном для восприятия человеком, используются методы `object.to_int`, `object.to_str`, `object.to_ary`, `object.to_sym`, `object.to_regexp`, `object.to_proc`, `object.to_hash` и т.д.

Для явного приведения типов используются частные методы экземпляров из модуля Kernel. Когда приведение типов невозможно возвращается nil.

.Array(object) *# -> array*

1. `object.to_ary`
2. `object.to_a`

.Float(object) *# -> float*

1. `object.to_f`

.Integer(object, numeral_system = 10) *# -> integer*

1. `object.to_int`
2. `object.to_i`

Когда первым аргументом передается текст, второй аргумент считается системой счисления (от 2 до 36). Для двоичной и шестнадцатеричной систем допускаются приставки 0b (0B) и 0x (0X). В другом случае передаваемый текст должен содержать только десятичные цифры.

.String(object) *# -> string*

1. `object.to_s`

.Hash(object) *# -> hash [Ruby 2.0]*

1. `object.to_hash`

Для nil или пустого массива возвращается `{}`.

```
Hash([])          # -> {}
Hash(nil)         # -> {}
Hash(key: :value) # -> {:key => :value}
Hash( [1, 2, 3] ) # -> TypeError!
```

.format(format, *object) *# -> string*

Синонимы: `sprintf`

Используется для получения форматированного текста `format % [*object]`.

9.3.2. Сравнение объектов

Проверка равенства

.==(object) *# -> [Object]*

Синонимы: `===`

Проверка равенства с приведением типов.

```
1 == 1.0 -> true
```

.eql?(object) *# -> [Object]*

Проверка равенства без приведения типов.

```
1.eql? 1.0 -> false
```

.equal?(object) *# -> [Object]*

Проверка идентичности двух ссылок.

```
1.equal? 1.0 -> false
"1".equal? "1" -> false
1.equal? 1 -> true
:a.equal? :a -> true
```

Comparable

В модуле определены операторы для проверки отношения и равенства объектов (`<`, `<=`, `>`, `>=`, `==`), основанные на работе оператора `.<=>`.

.between?(first, last)

Проверка входит ли объект между двумя заданными границами.

9.3.3. Копирование объектов

Создание копий

Так как все аргументы передаются по ссылке, то изменение их в теле метода может привести к непредсказуемым последствиям. Для решения этой проблемы перед использованием аргумента обычно создают его копию.

Свойства копии будут ссылаться на те же объекты, что и свойства оригинала.

```
a = { t1: 'sa ma ra', t2: 'john' }
b = a.clone

b[:t2] = 'jack'
b[:t1].sub!('ma', 'ha')

a[:t1] == 'sa ha ra' # -> true
```

.clone *# -> object [Object]*

Используется для создания копии объекта, сохраняющей все его модификаторы.

.dup *# -> object [Object]*

Используется для создания копии объекта, разрешенной к изменению.

.initialize_clone(object) *# -> object*

Метод вызывается при попытке создать копию объекта, сохраняющую все его модификаторы (для клона). В качестве аргумента передается клонируемый объект.

Во второй версии Ruby метод относится к частным.

.initialize_dup(object) *# -> object*

Метод вызывается при попытке создать копию объекта, разрешенной к изменению (для клона). В качестве аргумента передается клонируемый объект.

Во второй версии Ruby метод относится к частным.

.initialize_copy(object) *# -> object*

Метод вызывается при попытке создать копию объекта (для клона). В качестве аргумента передается клонируемый объект.

Псевдокод

Приведенный ниже псевдокод отражает выполняемые действия при клонировании объекта.

```
class Object
  def clone
    clone = self.class.allocate

    clone.copy_instance_variables(self)
    clone.copy_singleton_class(self)

    clone.initialize_clone(self)
    clone.freeze if frozen?

    clone
  end

  def dup
    dup = self.class.allocate
    dup.copy_instance_variables(self)
    dup.initialize_dup(self)
    dup
  end

  def initialize_clone(other)
    initialize_copy(other)
  end

  def initialize_dup(other)
    initialize_copy(other)
  end

  def initialize_copy(other)
    # ...
  end
end
```

Маршализация (Marshal)

Маршализация позволяет сохранять произвольные объекты, извлекать их из выполняемой программы и восстанавливать в другой программе (программы должны использовать одну версию интерпретатора).

Константы:

Marshal::MAJOR_VERSION - основная версия интерпретатора;
Marshal::MINOR_VERSION - дополнительная версия интерпретатора.

::dup(object, io = nil, deep = -1) *# -> string*

Используется для маршализации объекта. Принимается глубина вложенности маршализуемых свойств (по умолчанию значения свойств не сохраняются). Если передается поток, то результат будет записан в поток.

Невозможна маршализация:

- анонимных модулей или классов;
- объектов, связанных с ОС (файлы, каталоги, потоки и т.д);
- экземпляров MatchData, Data, Method, UnboundMethod, Proc, Thread, ThreadGroup, Continuation;
- объектов, определяющих собственные методы.

```
Marshal.dump ?$ # -> "\x04\bI\"\x06$\x06:\x06ET"
```

::load(marshal_data) *# -> object*

```
(marshal_data) { |result| } # -> object
```

Синонимы: `restore`

Используется для восстановления маршализованного объекта.

```
Marshal.load Marshal.dump(?$) # -> "$"
```

Глава 10

Подпрограммы

Подпрограмма - это именованный фрагмент кода, содержащий описание определенного набора действий. К подпрограммам относятся замыкания (процедуры и функции), методы и сопрограммы.

10.1. Замыкания (Proc)

Замыкание (англ. closure) в программировании - это процедура или функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции и не в качестве её параметров (а в окружающем коде). Эти переменные продолжают существовать во время выполнения функции, даже если во внешнем коде их уже не существует.

Замыкания создаются заново каждый раз в момент выполнения.

Замыкания, так же как и объекты служат для инкапсуляции функциональности и данных.

Замыкания могут быть переданы методам как обычные блоки. Для этого перед аргументом используется амперсанд.

Псевдопеременная self сохраняет значение, существовавшее в момент создания замыкания.

```
class FirstKlass
  def self.first; 1; end

  def first; 2; end

  @@first = proc { first }

  def klass; @@first.call; end

  def instance; instance_exec &@@first; end
end

FirstKlass.new.klass # -> 1
FirstKlass.new.instance # -> 2
```

Замыкания разделяются на процедуры (ведущие себя как блоки, но в действительности также относящиеся к функциям) и лямбда-функции (ведущие себя как методы).

10.1.1. Процедуры

Инструкция return в теле процедуры приводит к завершению выполнения метода, в теле которого объект был создан. Если выполнение метода уже завершено, то вызывается исключение `LocalJumpError`.

Процедуры принимают аргументы также как и обычные блоки.

::new { |*params| } *# -> proc*

Используется для создания нового замыкания.

```
def proc_from
  Proc.new
end

proc = proc_from { "hello" }
proc.call # -> "hello"
```

.proc { |*params| } *# -> proc [PRIVATE: Kernel]*

Версия предыдущего метода из модуля Kernel.

```
def foo
  f = Proc.new { return "return from foo from inside proc" }
  f.call # после вызова функции замыкания f осуществляется выход из foo
  # результатом работы функции foo является результат работы f замыкания
  return "return from foo"
end

puts foo # печатает "return from foo from inside proc"
```

10.1.2. Лямбда-функции

Инструкция return в теле лямбда-функций приводит к завершению выполнения функции.

Лямбда-функции принимают аргументы также как и обычные методы.

Для создания лямбда функций существует специальный синтаксис. Круглые скобки не обязательны. Параметры могут иметь значения по умолчанию.

```
->(*params) { } # -> lambda
```

.lambda { |*params| } *# -> lambda*

Используется для создания объекта.

```
def bar
  f = lambda { return "return from lambda" }
  f.call # после вызова функции замыкания f продолжается выполнение bar
  return "return from bar"
end

puts bar # печатает "return from bar"
```

10.1.3. Использование замыканий

Приведение типов

.to_proc *# -> self*

.to_s *# -> string*

Информация об объекте.

```
Proc.new { }.to_s # -> "#<Proc:0x8850f18@(irb):31>"
```

Операторы

.==(proc)

Синонимы: `eql?`

Проверка на равенство. Два замыкания равны, если относятся к копиям одного и того же объекта.

```
proc {} == proc {} # -> true
```

Во второй версии Ruby этот метод удален. Два замыкания считаются эквивалентными, только если ссылаются на один объект.

```
proc {} == proc {} # -> false
```

.===(arg) *# -> object*

Используется для выполнения замыкания.

```
proc { |a| a } === 1 # -> 1
```

Выполнение замыкания

.call(*arg) *# -> object*

Синонимы: `yield, .(*arg), [*arg]`

Используется для выполнения замыкания.

```
-> x {x**2}.(5) # -> 25
```

.curry(count = nil)*arg *# -> proc*

Используется для подготовки замыкания. Замыкание будет выполнено, когда передается достаточное количество аргументов. В другом случае замыкание сохраняет информацию о переданных аргументах.

Дополнительный аргумент ограничивает количество передаваемых аргументов (остальным параметрам присваивается nil).

```
b = proc { | x, y, z | x + y + z }
b.curry[1][2][3] # -> 6
b.curry[1, 2][3, 4] # -> 6
b.curry[1][2][3][4][5] # -> 0
b.curry(5) [ 1, 2 ][ 3, 4][5] # -> 6
b.curry(1) [1] # -> type_error!

b = lambda { | x, y, z | x + y + z }
b.curry[1][2][3] # -> 6
b.curry[1, 2][3, 4] # -> argument_error!
b.curry(5) # -> argument_error!
b.curry(1) # -> argument_error!
```

Остальное

.arity *# -> integer*

Количество принимаемых аргументов. Для произвольного количества возвращается отрицательное число. Его инверсия с помощью оператора ~ в результате возвращает количество обязательных аргументов.

```
proc {}.arity # -> 0
proc { || }.arity # -> 0
proc { |a| }.arity # -> 1
proc { |a, b| }.arity # -> 2
proc { |a, b, c| }.arity # -> 3
proc { |*a| }.arity # -> -1
proc { |a, *b| }.arity # -> -2
proc { | a, *b, c | }.arity # -> -3
```

.parameters *# -> array*

Информация о параметрах.

```
proc = lambda { | x, y = 42, *other | }
proc.parameters
# -> [ [:req, :x],  [:opt, :y], [:rest, :other] ]
```

.binding *# -> binding*

Используется для получения состояния выполнения программы для замыкания.

.lambda? *# -> bool*

Проверка относится ли объект к лямбда-функциям.

```
proc {}.lambda? # -> false
```

.source_location *# -> array*

Местоположение создания замыкания в виде `[filename, line]`. Для замыканий, создаваемых не на Ruby, возвращается nil.

```
proc {}.source_location # -> ["(irb)", 19]
```

.hash *# -> integer*

Цифровой код объекта.

```
proc {}.hash # -> -259341767
```

Мемоизация

Мемоизация - это хранение ранее вычисленного результата с помощью замыканий и вложенных подпрограмм.

```
closure = proc { counter = 0; proc {counter += 1} }.call
closure.call # -> 1
closure.call # -> 2
closure.call # -> 3
```

10.2. Методы

Подпрограммы могут быть созданы на основе уже существующих методов с помощью классов Method и UnboundMethod.

10.2.1. Method

Экземпляры класса сохраняют информацию о методе вместе с объектом, для которого он вызывается.

.method(name) *# -> method*

Используется для сохранения информации о переданном методе объекта. Отсутствие метода считается исключением.

```
12.method(?+) # -> #<Method: Fixnum#+>
```

.public_method(name) *# -> method*

Версия предыдущего метода для поиска только общих методов.

```
12.public_method(?+) # -> #<Method: Fixnum#+>
```

Операторы

.==(method)

Синонимы: `eql?`

Проверка на равенство. Объекты равны, если связаны с одним и тем же объектом и содержат информацию об одном и том же методе.

```
12.method(?+) == 13.method(?+) # -> false
```

Приведение типов

.to_proc *# -> lambda*

Используется для создания лямбда-функции на основе метода.

```
12.method(?+).to_proc # -> #<Proc:0x88cdd60 (lambda)>
```

.to_s *# -> string*

Информация об объекте.

```
12.method(?+).to_s # -> "#<Method: Fixnum#+>"
```

.unbind *# -> umethod*

Используется для удаления информации об объекте, для которого вызывается метод.

```
12.method(?+).unbind -> #<UnboundMethod: Fixnum#+>
```

Вызов метода

.call(*arg) *# -> object*

Используется для вызова метода с переданными аргументами.

```
12.method(?+).call 3 # -> 15
```

Остальное

.arity *# -> integer*

Количество принимаемых аргументов. Для произвольного количества возвращается отрицательное число. Его инверсия с помощью оператора ~ в результате возвращает количество обязательных аргументов. Для методов, определенных без помощи Ruby возвращается -1.

```
12.method(?+).arity # -> 1
```

.name *# -> symbol*

Идентификатор метода.

```
12.method(?+).name # -> :+
```

.owner *# -> module*

Модуль, в котором объявлен метод.

```
12.method(?+).owner # -> Fixnum
```

.parameters *# -> array*

Массив параметров метода.

```
12.method(?+).parameters # -> [ [:req] ]
```

.receiver *# -> object*

Объект, для которого метод вызывается. `12.method(?+).receiver # -> 12`

.source_location *# -> array*

Местоположение объявления метода в виде массива `[filename, line]`. Для методов, определенных без помощи Ruby, возвращается nil.

```
12.method(?+).source_location # -> nil
```

.hash *# -> integer*

Цифровой код объекта.

```
12.method(?+).hash # -> -347045594
```

10.2.2. UnboundMethod

Экземпляры класса сохраняют информацию только о методе.

.instance_method(name) *# -> umethod [Module]*

Используется для хранения информации о переданном методе. Отсутствие метода считается ошибкой.

```
Math.instance_method :sqrt # -> #<UnboundMethod: Math#sqrt>
```

.public_instance_method(name) *# -> umethod [Module]*

Версия предыдущего метода для поиска только общих методов.

```
Math.public_instance_method :sqrt # -> error
```

Операторы

.==(umethod)

Синонимы: `eql?`

Проверка на равенство. Объекты равны, если содержат информацию об одном и том же методе.

```
Math.instance_method(:sin) == Math.instance_method(:sin) # -> true
```

Приведение типов

.to_s *# -> string* Синонимы: `inspect`

Информация об объекте.

```
Math.instance_method(:sqrt).to_s # -> "#<UnboundMethod: Math#sqrt>"
```

.bind(object) *# -> method*

Используется для добавления информации об объекте, для которого метод вызывается. Если такая информация уже существовала, то объекты должны принадлежать к одному классу.

```
12.method(?+).unbind.bind 1 # -> #<Method: Fixnum#+>
12.method(?+).unbind.bind 1.0 # -> error!
```

Остальное

.arity *# -> integer*

Количество принимаемых аргументов. Для произвольного количества возвращается отрицательное число. Его инверсия с помощью оператора ~ в результате возвращает количество обязательных аргументов. Для методов, определенных без помощи Ruby возвращается -1.

```
Math.instance_method(:sqrt).arity # -> 1
```

.name *# -> symbol*

Идентификатор метода.

```
Math.instance_method(:sqrt).name # -> :sqrt
```

.owner *# -> module*

Модуль, в котором объявлен метод.

```
Math.instance_method(:sqrt).owner # -> Math
```

.parameters *# -> array*

Массив параметров метода.

```
Math.instance_method(:sqrt).parameters # -> [ [:req] ]
```

.source_location *# -> array*

Местоположение объявления метода в виде массива `[filename, line]`. Для методов, определенных без помощи Ruby, возвращается nil.

```
Math.instance_method(:sqrt).source_location # -> nil
```

.hash *# -> integer*

Цифровой код объекта.

```
Math.instance_method(:sqrt).hash # -> 563385534
```

10.3. Сопрограммы (Fiber)

Сопрограмма - это фрагмент кода, поддерживающий несколько входных точек и остановку или продолжение выполнения с сохранением состояния выполнения. Сопрограмму также можно рассматривать как поток выполнения, прерываемый специальной инструкцией.

Сопрограммы также могут использоваться для реализации многопоточности на уровне программы (в действительности код выполняется в единственном потоке выполнения). Такие потоки также называют ”green thread”. Использование сопрограмм позволяет уменьшить накладные расходы на переключение и обмен данными, так как не требует взаимодействия с ядром ОС.

Проблема при использовании сопрограмм в том, что выполнение системного вызова будет блокировать процесс выполнения программы. Сопрограммы должны использовать специальные методы ввода/вывода, не блокирующие процесс выполнения. Также стоит заметить что управление переключением сопрограмм выполняется вручную и требует дополнительных затрат при разработке.

Для управления сопрограммами создаются контрольные точки с помощью метода `::yield` и осуществляется последовательный переход между ними с помощью метода `.resume`.

::new { |*params| } *# -> fiber*

Используется для создания сопрограммы. Блок при этом не выполняется.

::yield(*temp_result) *# -> object*

Используется для создания контрольной точки. Переданные аргументы возвращаются в результате вызова `.resume`.

Возвращаются объекты, переданные при вызове метода `.resume` или полученные в результате выполнения последнего выражения в теле сопрограммы.

.resume(*args) *# -> temp_result*

Используется для выполнения блока до следующей контрольной точки.

Если метод был вызван впервые, то переданные аргументы отправляются в сопрограмму. В другом случае они возвращаются в результате вызова метода `::yield` в теле блока.

Если сопрограмма уже выполнена, то вызывается исключение `FiberError`.

Глава 11

Псевдослучайные числа (Random)

Для генерации псевдослучайных чисел в Ruby используется алгоритм ”Вихрь Мерсена”, предложенный в 1997 году Мацумото и Нисимурой. Его достоинствами являются колоссальный период (`2 ** 19937 - 1`), равномерное распределение в 623 измерениях, быстрая генерация случайных чисел (в 2-3 раза быстрее, чем стандартные генераторы). Однако, существуют алгоритмы, распознающие последовательность, порождаемую ”Вихрем Мерсенна”, как неслучайную.

::new(seed = Random.new_seed) *# -> random*

Используется для создания генератора. Объект, переданный методу, необходим для исключения повторяющихся чисел. Его также называют ”соль”.

```
Random.new # -> #<Random:0xa0a1fa4>
```

::rand(number = 0) *# -> random_number*

Используется для получения случайного числа.

Проверяется результат `number.to_i.abs`:

- когда он равен нулю или ссылается на nil, то возвращается псевдослучайная десятичная дробь в диапазоне `0.0...1.0`.
- в другом случае возвращается псевдослучайное число в диапазоне `0...number.to_i.abs`.

```
Random.rand # -> 0.8736231696463861
```

.rand(number = 0) *# -> random_number [PRIVATE: Kernel]*

Версия предыдущего метода из модуля Kernel.

::new_seed *# -> integer*

Используется для вычисления новой ”соли”.

```
Random.new_seed # -> 69960780063826734370396971659065074316
```

::srand(number = 0) *# -> number*

Используется для вычисления новой ”соли”. Когда аргумент равен нулю, то используется время вызова, идентификатор процесса и порядковый номер вызова. С помощью аргумента программа может быть детерминирована во время тестирования. Возвращается предыдущее значение.

.srand(number = 0) *# -> number [PRIVATE: Kernel]*

Версия предыдущего метода из модуля Kernel.

11.1. Генераторы

.bytes(bytesize) *# -> string*

Используется для получения случайного двоичного текста.

```
Random.new.bytes 2 -> "\xA1W"
```

.rand(object = nil) *# -> number*

- Когда передано целое число, возвращается псевдослучайное число в диапазоне `0...object`;
- Передача отрицательного числа или нуля считается исключением;
- Когда передана десятичная дробь, возвращается псевдослучайная десятичная дробь в диапазоне `0.0...object`;
- Когда передан диапазон, возвращается случайный элемент диапазона. Для начальной и конечной границ должны быть определены операторы - (разность) и + (сумма);
- Остальные случаи считаются исключением.

.seed *# -> integer*

Используется для получения новой ”соли”.

```
Random.new.seed # -> 173038287409845379387953855893202182131
```

Глава 12

Дата и время

12.1. Время (Time)

Добавленные модули: Comparable

Экземпляры класса Time - это абстрактные объекты, содержащие информацию о времени. Время хранится в секундах, начиная с 01.01.1970 00:00 UTC. Системы отсчета времени GMT (время по Гринвичу) и UTC (универсальное время) трактуются как эквивалентные.

При сравнении разных объектов необходимо помнить, что различные пояса могут иметь смещения по времени от UTC.

Аргументы:

year - год;
month - месяц: либо целое число от 1 до 12, либо текст, содержащий первые три буквы английского названия месяца (аббревиатуру);
day - номер дня в месяце: целое число от 1 до 31;
wday - номер дня в неделе: целое число от 0 до 6, начиная с воскресенья;
yday - номер дня в году: целое число от 1 до 366;
isdst - летнее время: логическая величина;
zone - временная зона: текст;
hour - час: целое число от 0 до 23;
min - минуты: целое число от 0 до 59;
sec - секунды: целое число или десятичная дробь от 0 до 60;
usec - микросекунды: целое число или десятичная дробь от 0 до 999.

12.1.1. Создание объекта

::new *# -> time*

```
(year, month = 1, day = 1, hour = 0, min = 0, sec = 0, usec = 0, zone)
# -> time
```

Используется для создания объекта. При вызове без аргументов возвращается текущее системное время. Последним аргументом передается смещение относительно UTC в виде текста ”+00:00” или количества секунд. По умолчанию берется системное смещение часового пояса.

```
Time.new 1990, 3, 31, nil, nil, nil, "+04:00"
# -> 1990-03-31 00:00:00 +0400
```

::now *# -> time*

Используется для получения текущего системного времени.

```
Time.now -> 2011-09-17 10:36:26 +0400
```

::at(time) *# -> time*

```
( sec, usec = nil ) # -> time
```

Используется для создания объекта. Принимаются секунды и микросекунды, прошедшие с начала точки отсчета UTC. Время вычисляется с учетом смещения часового пояса.

```
Time.at 1 -> 1970-01-01 03:00:01 +0300
```

::utc(year, month = 1, day = 1, hour = 0, min = 0, sec = 0, usec = 0)

```
( sec, min, hour, day, month, year, wday, yday, isdst ) # -> time
```

Синонимы: `gm`

Используется для создания объекта.

```
Time.utc 1990, 3, 31 -> 1990-03-31 00:00:00 UTC
```

::local(year, month = 1, day = 1, hour = 0, min = 0, sec = 0, usec = 0, zone)

```
(sec, min, hour, day, month, year, wday, yday, isdst, zone) # -> time
```

Синонимы: `time`

Используется для создания объекта, с учетом смещения часового пояса.

```
Time.local 1990, 3, 31 -> 1990-03-31 00:00:00 +0400
```

12.1.2. Приведение типов

.to_s *# -> string*

Синонимы: `inspect`

Преобразование в текст.

```
Time.local( 1990, 3, 31 ).to_s -> "1990-03-31 00:00:00 +0400"
```

.to_a *# -> array*

Индексный массив вида:

```
[ self.sec, self.min, self.hour,
  self.day, self.month, self.year,
  self.wday, self.yday,
  self.isdst, self.zone ]
```

```
Time.local( 1990, 3, 31 ).to_a
# -> [ 0, 0, 0, 31, 3, 1990, 6, 90, true, "MSD" ]
```

.to_i *# -> integer*

Синонимы: `tv_sec`

Количество секунд прошедших начиная с 1970-01-01 00:00:00 UTC.

```
Time.local( 1990, 3, 31 ).to_i # -> 638827200
```

.to_r *# -> rational*

Количество секунд прошедших начиная с 1970-01-01 00:00:00 UTC в виде рациональной дроби.

```
Time.local( 1990, 3, 31 ).to_r # -> (638827200/1)
```

.to_f *# -> float*

Количество секунд прошедших начиная с 1970-01-01 00:00:00 UTC в виде десятичной дроби.

```
Time.local( 1990, 3, 31 ).to_f # -> 638827200.0
```

12.1.3. Операторы

.+(number) *# -> time*

Используется для прибавления секунд.

```
Time.local( 1990, 3, 31 ) + 3600 # -> 1990-03-31 01:00:00 +0400
```

.-(time) *# -> float*

Разница в секундах.

```
Time.local( 1990, 3, 31 ) - Time.new( 1990, 3, 31 ) -> 0.0
```

.-(number) *# -> time*

Используется для уменьшения секунд.

```
Time.local( 1990, 3, 31 ) - 3600 -> 1990-03-30 23:00:00 +0400
```

.<=>(time)

Сравнение.

```
Time.local( 1990, 3, 31 ) <=> Time.new( 1990, 3, 31 ) -> 0
```

12.1.4. Форматирование

.strftime(format) *# -> string*

Используется для форматирования времени на основе переданной форматной строки.

12.1.5. Системы остчета

.getutc *# -> time*

Синонимы: `getgm`

Время относительно UTC (без смещения часовых поясов).

```
Time.local( 1990, 3, 31 ).getutc # -> 1990-03-30 20:00:00 UTC
```

.utc *# -> self*

Синонимы: `gmtime`

Версия предыдущего метода, изменяющая значение объекта.

.getlocal(zone = nil) *# -> time*

Время с смещением часового пояса. Смещение может быть явно передано методу (по умолчанию используется системное смещение).

```
Time.local(1990, 3, 31).getutc.getlocal # -> 1990-03-31 00:00:00 +0400
```

.localtime(zone = nil) *# -> self*

Версия предыдущего метода, изменяющая значение объекта.

12.1.6. Статистика

.asctime *# -> string*

Синонимы: `ctime`

Время в виде текста.

```
Time.local( 1990, 3, 31 ).asctime # -> "Sat Mar 31 00:00:00 1990"
```

.utc_offset *# -> integer*

Синонимы: `gmt_offset, gmtoff`

Смещение часового пояса относительно UTC в секундах.

```
Time.local( 1990, 3, 31 ).utc_offset # -> 14400
```

.zone *# -> string*

Название временной зоны.

```
verb!Time.local( 1990, 3, 31 ).zone # -> "MSD"
```

.year *# -> integer*

Номер года.

```
Time.local( 1990, 3, 31 ).year # -> 1990
```

.month *# -> integer*

Синонимы: `mon`

Номер месяца.

```
verb!Time.local( 1990, 3, 31 ).month # -> 3
```

.yday *# -> integer*

Номер дня в году от 1 до 366.

```
Time.local( 1990, 3, 31 ).yday # -> 90
```

.day *# -> integer*

Синонимы: `mday`

Номер дня в месяце.

```
Time.local( 1990, 3, 31 ).day # -> 31
```

.wday *# -> integer*

Номер дня недели (от 0 до 6, начиная с воскресенья).

```
Time.local( 1990, 3, 31 ).wday # -> 6
```

.hour *# -> integer*

Час дня (число от 0 до 23).

```
Time.local( 1990, 3, 31 ).hour # -> 0
```

.min *# -> integer*

Количество минут (число от 0 до 59).

```
Time.local( 1990, 3, 31 ).min # -> 0
```

.sec *# -> integer*

Количество секунд (число от 0 до 60).

```
Time.local( 1990, 3, 31 ).sec # -> 0
```

.subsec *# -> integer*

Дробная часть секунд.

```
Time.local( 1990, 3, 31 ).subsec # -> 0
```

.usec *# -> integer*

Синонимы: `tv_usec`

Количество микросекунд.

```
Time.local( 1990, 3, 31 ).usec # -> 0
```

.nsec *# -> integer*

Синонимы: `tv_nsec`

Количество наносекунд.

```
Time.local( 1990, 3, 31 ).nsec # -> 0
```

12.1.7. Предикаты

.monday? *# -> bool*

Проверка является ли понедельник днем недели.

```
Time.local( 1990, 3, 31 ).monday? # -> false
```

.tuesday? *# -> bool*

Проверка является ли вторник днем недели.

```
Time.local( 1990, 3, 31 ).tuesday? # -> false
```

.wednesday? *# -> bool*

Проверка является ли среда днем недели.

```
Time.local( 1990, 3, 31 ).wednesday? # -> false
```

.thursday? *# -> bool*

Проверка является ли четверг днем недели.

```
Time.local( 1990, 3, 31 ).saturday? # -> false
```

.friday? *# -> bool*

Проверка является ли пятница днем недели.

```
Time.local( 1990, 3, 31 ).friday? # -> false
```

.saturday? *# -> bool*

Проверка является ли суббота днем недели.

```
Time.local( 1990, 3, 31 ).saturday? # -> true
```

.sunday? *# -> bool*

Проверка является ли воскресенье днем недели.

```
Time.local( 1990, 3, 31 ).saturday? # -> false
```

.utc? *# -> bool*

Синонимы: `gmt?`

Проверка используется ли время относительно UTC.

```
Time.local( 1990, 3, 31 ).utc? # -> false
```

.dst? *# -> bool*

Синонимы: `isdst`

Проверка используется ли переход на летнее время.

```
Time.local( 1990, 3, 31 ).dst? # -> true
```

12.1.8. Остальное

.hash *# -> integer*

Цифровой код объекта.

```
Time.local( 1990, 3, 31 ).hash # -> -494674000
```

.round(precise = 0) *# -> time*

Используется для округления секунд с заданной точностью. Точность определяет размер дробной части.

```
Time.local( 1990, 3, 31 ).round 2 # -> 1990-03-31 00:00:00 +0400
```

Глава 13

Типы данных

13.1. Логические величины

Классы TrueClass, FalseClass и NilClass - это собственные классы объектов true, false и nil соответственно. Также существуют константы TRUE, FALSE, NIL, которые могут быть переопределены (но зачем?).

NilClass	FalseClass	TrueClass
nil & obj -> false	false & obj -> false	true & bool -> bool
nil ~ bool -> bool	false ~ bool -> bool	true ~ bool -> !bool
nil | bool -> bool	false | bool -> bool	true | object -> true
nil.to_s -> ""	false.to_s -> "false"	true.to_s -> "true"

```
nil.nil? # -> true
nil.inspect # -> "nil"
nil.rationalize # -> (0/1)
nil.to_r # -> (0/1)
nil.to_a # -> []
nil.to_c # -> (0+0i)
nil.to_f # -> 0.0
nil.to_i # -> 0
nil.to_h # -> {} [Ruby 2.0]
```

13.2. Symbol

Добавленные модули: Comparable

Большинство методов сначала преобразует объект в текст.

::all_symbols *# -> array* Массив всех существующих экземпляров класса.

13.2.1. Приведение типов

.to_sym *# -> symbol* Синонимы: `intern`

.to_s *# -> string*

Синонимы: `id2name`

Используется для получения текстовое значения.

```
:Ruby.to_s # -> "Ruby"
```

.inspect *# -> string*

Информация об объекте.

```
:Ruby.inspect # -> ":Ruby"
```

.to_proc *# -> proc*

Используется для получения подпрограммы, выполняющей то же, что и метод с аналогичным идентификатором.

```
1.next # -> 2
:next.to_proc.call(1) # -> 2
```

13.2.2. Операторы

```
symbol <=> object # -> symbol.to_s <=> object
symbol =~ object # -> symbol.to_s =~ object
symbol[*object] # -> symbol.to_s[*object]
symbol.slice(*object) # -> symbol.to_s.slice(*object)
```

13.2.3. Изменение регистра

.capitalize *# -> symbol* Выполняемое выражение: `self.to_s.capitalize.to_sym`.

.swapcase *# -> symbol* Выполняемое выражение: `self.to_s.swapcase.to_sym`.

.upcase *# -> symbol* Выполняемое выражение: `self.to_s.upcase.to_sym`.

.downcase *# -> symbol* Выполняемое выражение: `self.to_s.downcase.to_sym`.

13.2.4. Остальное

.encoding *# -> encoding*

Кодировка.

```
:Ruby.encoding # -> #<Encoding:US-ASCII>
```

.empty? *# -> bool* Выполняемое выражение: `self.to_s.empty?`.

.length *# -> integer*

Синонимы: `size`

Выполняемое выражение: `self.to_s.length` или `self.to_s.size`.

.casecmp(object) Выполняемое выражение: `self.to_s.casecmp(object)`.

.next *# -> symbol*

Синонимы: `succ`

Выполняемое выражение: `self.to_s.next.to_sym` или `self.to_s.succ.to_sym`

13.3. Структуры

Добавленные модули: Enumerable

Структура данных - это объект, имеющий только свойства. Иногда структуры также используют при необходимости передавать несколько аргументов.

Для создания структур используется класс Struct.

::new(name = nil, *attribute) *# -> class*

Используется для создания в теле Struct нового класса структур и объявления переданных свойств (ссылающихся на nil). Если название класса не передано, то создается анонимный класс. Анонимный класс получит идентификатор, только если будет присвоен константе.

Для нового класса определяется конструктор, принимающий значения для объявленных свойств.

```
Struct.new "Ключ", :объект # -> Struct::Ключ
# только для примера. Никогда так больше не делайте :)
Struct::Ключ.new [1, 2, 3]
# ->  #<struct Struct::Ключ объект=[1, 2, 3]>
```

13.3.1. Приведение типов

.to_s *# -> string*

Синонимы: `inspect`

Информация об объекте.

```
Struct::Ключ.new( [1, 2, 3] ).to_s
# -> "#<struct Struct::Ключ объект=[1, 2, 3]>"
```

.to_a *# -> array*

Синонимы: `values`

Массив значений свойств.

```
Struct::Ключ.new( [1, 2, 3] ).to_a # -> [ [1, 2, 3] ]
```

.to_h *# -> hash [Ruby 2.0]*

Массив свойств и их значений.

```
Customer = Struct.new :name, :address, :zip
joe = Customer.new "Joe Smith", "123 Maple, Anytown NC", 12345
joe.to_h[:address] # -> "123 Maple, Anytown NC"
```

13.3.2. Элементы

Элементами структур считаются свойства. Для доступа к свойствам определены операторы `[]` и `[]=`. Они принимают идентификаторы или индексы свойств. Индексация свойств начинается с 0 и соответствует порядку, использовавшемуся при создании структуры. Индекс может быть отрицательным (индексация, начинается с -1).

Использование несуществующего свойства считается исключением.

.[attr] *# -> object*

Используется для получения значения свойства.

```
Struct::Ключ.new( [1, 2, 3] )["объект"] # -> [1, 2, 3]
Struct::Ключ.new( [1, 2, 3] )[0] # -> [1, 2, 3]
Struct::Ключ.new( [1, 2, 3] )[-1] # -> [1, 2, 3]
```

.[attr]=(object) *# -> object*

Используется для изменения свойства.

```
Struct::Ключ.new( [1, 2, 3] )["объект"] = :array # -> :array
Struct::Ключ.new( [1, 2, 3] )[0] = :array # -> :array
Struct::Ключ.new( [1, 2, 3] )[-1] = :array # -> :array
```

.values_at(*integer) *# -> array*

Используется для получения массива значений свойств с переданными индексами. При вызове без аргументов возвращается пустой массив.

```
Struct::Ключ.new( [1, 2, 3] ).values_at 0, -1
# -> [ [1, 2, 3], [1, 2, 3] ]
```

13.3.3. Итераторы

.each { |object| } *# -> self*

Перебор значений свойств.

.each_pair { | name, object | } *# -> self*

Перебор идентификаторов свойств вместе с значениями.

13.3.4. Остальное

.hash *# -> integer*

Цифровой код объекта.

```
Struct::Ключ.new( [1, 2, 3] ).hash # -> -764829164
```

.size *# -> integer*

Синонимы: `length`

Количество свойств.

```
Struct::Ключ.new( [1, 2, 3] ).size # -> 1
```

.members *# -> array*

Массив идентификаторов свойств.

```
Struct::Ключ.new( [1, 2, 3] ).members # -> [:объект]
```

Часть III

Работа программы

Глава 14

Выполнение программы

14.1. Запуск программы

Запуск программ выполняется из терминала. Общий вид команды:

```
ruby [keys] [path] [args]
```

- *keys* - заранее определенные спецсимволы или идентификаторы, влияющие на выполнение программы;

  path - путь к запускаемой программе. Поиск программы выполняется относительно текущего каталога.

  Если имя программы не указано, или передается одиночный дефис, то интерпретатор будет выполнять код, полученный из стандартного потока для чтения информации (обычно связанного с терминалом).

  args - произвольный набор символов, которые будут переданы программе как элементы индексного массива ARGV.

Файлы, доступные для выполнения (скрипты или исполняемые файлы), могут содержать информацию об интерпретаторе в первом комментарии (shebang) программы:

```
#!/usr/bin/env ruby [keys] [args]
```

В этом случае для запуска программы требуется только ввести в терминале путь к ней.

Кроме переданных ключей на выполнение программы также могут влиять переменные окружения и предопределенные глобальные переменные.

Переменные окружения - это переменные, установленные операционной системой. В Linux список установленных переменных может быть получен с помощью команды `env`.

14.2. Ход выполнения

Ход выполнения программы - это непосредственное выполнение ее кода.

Ход выполнения может быть разделен на три этапа: подготовка, выполнение и завершение выполнения.

BEGIN и END - это предложения, выполняющиеся на стадиях подготовки и завершения выполнения соответственно. Тело каждого предложения определяет собственную локальную область видимости и выполняется строго один раз.

- *BEGIN* - код выполняется на стадии подготовки выполнения. Если в коде программы используется несколько таких предложений, то они выполняются последовательно в порядке записи;

  END - код выполняется на стадии завершения выполнения. Если в коде программы используется несколько таких предложений, то они выполняются последовательно в обратном порядке. Для выполнения этого предложения используется экземпляр File, на который ссылается константа DATA;

  __END__ - аналогично END;

  at_exit - аналогично END.

14.3. Завершение выполнения

Для завершения выполнения программы используются частные методы экземпляров из модуля Kernel (методы влияют на любой поток выполнения, в теле которого взываются).

.sleep(sec = nil) *# -> sec*

Используется для временной остановки выполнения (по умолчанию навсегда). В результате возвращается время фактического ожидания.

.exit(state = true)

Используется для завершения выполнения с помощью исключения `SystemExit`.

.exit!(state = false)

Используется для немедленного завершения выполнения.

.abort(mesage = nil)

Используется для немедленного завершения выполнения. Аргумент записывается в стандартный поток для вывода ошибок. Аналогично выполнению `exit false`.

14.4. Вызов системных команд

На Ruby довольно часто создают небольшие скрипты, облегчающие вызов различных системных команд. Для этого используются частные методы экземпляров из модуля Kernel.

.‘(code) *# -> string*

Используется для вызова системной команды.

```
ruby --help
```

Тот же эффект достигается при ограничении текста произвольными разделителями с использованием приставки `%x` (`%x[ruby --help]`).

.exec(env, command, options)

Используется для замены текущего процесса выполнения на выполнение системной команды. Невозможность выполнения команды считается исключением.

Во второй версии Ruby нестандартные файловые дескрипторы закрываются автоматически.

Аргументы:

env (hash): управление переменными окружения.

- *name:* значение для переменной окружения;
  name: nil, удаление переменной окружения.

command: системный вызов.

- *string* - текст команды для используемой оболочки: по умолчанию в Unix - это `"/bin/sh"`, а в Windows - `ENV["RUBYSHELL"]` или `ENV["COMSPEC"]`;
  *string, *arg* - текст команды и передаваемые аргументы;
  *[string, first_arg], *arg* - текст команды, первый аргумент и остальные аргументы.

option (hash): дополнительный аргумент.

- *unsetenv_others:* true, удаление всех переменных окружения, кроме переданных методу;
  pgroup: группировка процессов:
  - true для создания новой группы;
  - integer для сохранения процесса в соответствующей группе;
  - nil для отмены группировки.

  chdir: путь к текущему рабочему каталогу;
  umask: права доступа для создаваемых файлов или каталогов.

.syscall(number, *args)

Используется для выполнения системного вызова с переданным цифровым идентификатором (для Unix систем идентификаторы и функции описаны в файле syscall.h).

Дополнительно (не более девяти аргументов) методу передаются либо текст, содержащий указатель на последовательность байт, либо размер указателя в битах.

Невозможность выполнения системного вызова считается исключением `SystemCallEror`.

Невозможность вызова метода считается исключением `NotImplementedError`.

Метод непереносим и небезопасен в использовании.

Глава 15

Чтение и запись данных

15.1. IO (потоки)

Добавленные модули: Enumerable и File::Constants

Поток - это абстракция, используемая для ввода и вывода данных в единой манере. Вывод также называют чтением (извлечением) данных, а ввод - записью (передачей) данных.

Потоки являются удобным унифицированным программным интерфейсом для чтения или записи файлов (в том числе специальных и, в частности, связанных с устройствами), сокетов и передачи данных между процессами.

Модели ввода/вывода:

- *блокирующая:* процесс выполнения блокируется до тех пор пока ответ системы не будет записан в буфер приложения. Часто используется в отдельных процессах или потоках выполнения. К сожалению накладные расходы на создание процессов или потоков достаточно высоки.

  неблокирующая: процесс выполнения не блокируется. При попытке получить данные до их записи в буфер, приложению отправляется сигнал. На основе обработки сигналов создается механизм опроса (polling) потоков. Опрос нескольких потоков приводит к большим накладным расходам.

  мультиплексирование: опрос нескольких потоков в цикле. Возвращается первый доступный поток.

  асинхронная: ядру дается команда начать операцию и уведомить приложение когда операция будет полностью завершена (включая запись в буфер приложения).

15.1.1. Управление потоками

Виды потоков описаны в приложении.

Открытие потока

Для открытия потока требуется объект, с которым поток будет связан. Обычно это файл, но подойдет и экземпляр любого класса, производного от IO.

::new(file, mode, options = nil) *# -> io*

Синонимы: `for_fd`

Используется для открытия нового потока, связанного с переданным файлом. Создаваемый поток не может получить права доступа, отсутствующие при открытии файла.

::open(file, mode, options = nil) *# -> io*

```
( file, mode, options = nil ) { |io| } # -> object
```

Версия предыдущего метода, позволяющая работать с потоком в теле блока (поток автоматически закрывается после завершения выполнения блока).

::sysopen(path, mode = nil, *perm) *# -> integer*

Используется для низкоуровневого открытия файла. Возвращается файловый дескриптор.

::copy_stream(old_io, new_io, start = nil, syze = nil) *# -> integer*

Используется для копирования содержимого потока. Возвращается количество скопированных байт. Дополнительные аргументы определяют начало и размер копируемого фрагмента. Если переданный размер превышает размер потока, то копирование выполняется до последнего байта.

Модификация потока

.reopen(io) *# -> self*

```
( path, mode ) # -> self
```

Используется для обновления потока.

.binmode *# -> self*

Используется для установки двоичного режима.

.autoclose=(bool) *# -> bool*

Используется для установки режима автоматического закрытия потока.

.close_on_exec=(bool) *# -> nil*

Метод используется чтобы запретить использование текущего файлового дескриптора в производных процессах выполнения.

Во второй версии Ruby ограничение по умолчанию устанавливается для всех дескрипторов. Чтобы передать файловый дескриптор другому процессу предполагается использование метода `.spawn`.

.advise(symbol, start = 0, size = 0) *# -> nil*

Используется для оптимизации работы с файлом. Дополнительные аргументы ограничивают фрагмент файла, с которым ведется работа.

- *:normal* - обычная работа (по умолчанию);
  :sequintial - последовательный доступ к данным; *:random* - произвольный доступ к данным;
  :willneed - работа с файлом в ближайшем будущем;
  :dontneed - работа с файлом в ближайшем будущем выполняться не будет;
  :noreuse - работа с файлом будет выполняться только один раз.

.fcntl(integer, object) *# -> integer2*

Низкоуровневое управление потоком см. fcntl(2).

.ioctl(integer, object) *# -> integer2*

Низкоуровневое управление потоком см. ioctl(2).

Закрытие потока

Поток может быть закрыт автоматически сборщиком мусора.

.close *# -> nil*

Используется для закрытия потока и передачи всех изменений из буфера программы в ОС.

.close_read *# -> nil*

Используется для закрытия доступа на чтение. Невозможность операции (поток не связан с конвейером) считается исключением.

.close_write *# -> nil*

Используется для закрытия доступа для записи. Невозможность операции (поток не связан с конвейером) считается исключением.

Кодировка

.external_encoding *# -> encoding*

Внешняя кодировка. Если поток доступен для записи, но кодировка при этом не задана, то возвращается nil.

.internal_encoding *# -> encoding*

Внутренняя кодировка или nil.

.set_encoding(*encoding, options = nil) *# -> io*

Используется для изменения внешней и внутренней кодировок. Опции применяются при преобразовании кодировок.

Предикаты

.autoclose? *# -> bool*

Проверка автоматического закрытия потока.

.binmode? *# -> bool*

Проверка работы в двоичном режиме.

.eof? *# -> bool*

Синонимы: `eof`

Проверка достижения конца файла (поток должен быть доступен для чтения). Если поток не связан с конвейером или сокетом, то процесс выполнения программы блокируется до тех пор пока не закончатся действия с данными или поток не будет закрыт.

.tty? *# -> bool*

Синонимы: `isatty`

Проверка связи с терминалом. Если поток связан с терминалом, то вся информация, записанная в поток, будет отображаться в терминале. Данный предикат позволяет изменять формат вывода данных в зависимости от того, чем они будут отображены.

.closed? *# -> bool*

Проверка закрыт ли поток.

.close_on_exec? *# -> bool*

Метод используется для проверки доступа к файловому дескриптору из производного процесса.

```
f = open '/dev/null'
f.close_on_exec?                # -> false
f.close_on_exec = true
f.close_on_exec?                # -> true
f.close_on_exec = false
f.close_on_exec?                # -> false
```

Остальное

.stat *# -> a_file_stat*

Используется для получения информации об открытом файле.

15.1.2. Приведение типов

::try_convert(object) *# -> io*

Используется для преобразования аргумента в поток с помощью метода `.to_io`. Если такой метод не определен, то возвращается nil.

.to_i *# -> integer*

Синонимы: `fileno`

Дескриптор файла.

.to_io *# -> io*

15.1.3. Чтение данных

Поток должен быть доступен для чтения.
Дополнительные опции используются при открытии потока.

Опции:

encoding: внешняя кодировка. Игнорируется если указана размер фрагмента;
mode: вид создаваемого потока. Модификатор r обязателен;
open_args: массив аргументов, используемых при открытии файла.

Фрагменты

::read(path, start = nil, bytesize = nil, options = nil) *# -> string*

Используется для получения данных (по умолчанию всего файла). После чтения данных поток закрывается. Если размер требуемого фрагмента больше чем размер оставшихся данных, то извлекаются все данные.

::binread(path, start = nil, size = nil) *# -> string*

Используется для получения двоичных данных в режиме "rb:ASCII".

.read(bytesize = nil, buffer = nil) *# -> buffer*

Используется для получения данных из текущего потока (по умолчанию всех). Если размер фрагмента ограничен, то чтение выполняется в двоичном режиме. Дополнительный аргумент служит для хранения полученных данных.

- Если передается ноль, то возвращается пустой текст;
- Если в начале чтения достигнут конец файла, то возвращается ссылка nil (если размер фрагмента ограничен). В другом случае возвращается пустой текст.

.sysread(bytesize, buffer = nil) *# -> buffer*

Используется для получения данных с помощью низкоуровневые возможности системы.

.read_nonblock(bytesize, buffer = nil) *# -> buffer*

Используется для получения данных, не блокируя процесс выполнения, если система не готова выполнить запрос к файлу немедленно. Вместо этого программа принимает сигнал. Используется при реализации многопоточности на основе потоков выполнения или сопрограмм.

.readpartial(bytesize, buffer = nil) *# -> buffer*

Используется для получения данных из конвейеров, сокетов и терминалов, блокируя процесс выполнения:

- если буфер пуст;
- если поток пуст;
- если поток не достиг конца файла;

После блокировки ожидается получение данных или достижения конца файла. При получении данных они возвращаются, достижение конца файла считается исключением `EOFError`.

Процесс выполнения не блокируется, если поток содержит данные - в этом случае данные возвращаются.

Действие метода сходно с `io.sysread`, но, в отличии от него при возможности читает данные из буфера, вместо вызова исключения `IOError`. Также не вызываются `Errno::EWOULDBLOCK` и `Errno::EINTR`, а повторяется попытка чтения (метод игнорирует модификатор NONBLOCK).

Строки

::readlines(path, sep = $/, options = nil) *# -> array*

```
( path, size, options = nil ) # -> array
( path, sep, size, options = nil ) # -> array
```

Используется для сохранения строк в массиве (размер массива может быть ограничен). Символом перевода строки считается переданный разделитель (если передается пустой текст, то обрабатывается ”/n/n”).

.readlines(sep = $/, size = nil) *# -> array*

Версия метода для получения строк из текущего потока.

.gets(sep = $/, bytesize = nil) *# -> string*

Используется для последовательного извлечения строк из потока (размер строки может быть ограничен). Символом перевода строки считается переданный разделитель (если передается пустой текст, то обрабатывается ”/n/n”). Если достигнут конец файла, то возвращается nil.

Полученная в результате строка связывается с глобальной переменной `$_`.

.readline(sep = $/, bytesize = nil) *# -> string*

Версия предыдущего метода, считающая достижение конца файла исключением.

.lineno *# -> integer*

Позиция (порядковый номер) извлекаемой строки (`$.`).

.lineno=(pos) *# -> integer*

Используется для изменения позиции (порядкового номера) извлекаемой строки. Позиция обновляется при последующем чтении данных из потока.

.seek(offset, object = IO::SEEK_SET) *# -> 0*

Используется для изменения позиции (порядкового номера) извлекаемой строки относительно текущего положения и переданного смещения.

Константы:

- `IO::SEEK_CUR` -> новая_позиция = текущая_позиция + offset
  `IO::SEEK_END` -> новая_позиция = конец_файла + offset
  `IO::SEEK_SET` -> новая_позиция = offset

.sysseek(offset, object = IO::SEEK_SET) *# -> 0*

Версия предыдущего метода, использующая низкоуровневые возможности системы.

.rewind *# -> 0*

Используется для обнуления позиции (порядковый номер) извлекаемой строки. Метод не может быть вызван для потоков, связанных с конвейерами, терминалами или сокетами.

Символы

.getc *# -> string*

Используется для последовательного извлечения символов из потока. Если достигнут конец файла, возвращается nil.

.readchar *# -> string*

Версия предыдущего метода, считающая достижение конца файла исключением.

Байты

.getbyte *# -> integer*

Используется для последовательного извлечения байтов из потока. Если достигнут конец файла, возвращается nil.

.readbyte *# -> integer*

Версия предыдущего метода, считающая достижение конца файла исключением.

.pos *# -> integer*

Синонимы: `tell`

Позиция (порядковый номер) извлекаемого байта.

.pos=(pos) *# -> integer*

Используется для изменения позиции (порядкового номера) извлекаемого байта. Позиция обновляется при последующем чтении данных из потока.

Итераторы

::foreach(path, sep = $/, options = nil) { |string| } *# -> nil*

```
( path, size, options = nil ) { |string| } # -> nil
( path, sep, size, options = nil ) { |string| } # -> nil
```

Перебор строк (чтение выполняется с помощью `IO::readlines`).

.each(sep = $/) { |string| } *# -> self*

```
(size) { |string| } # -> self
( sep, size) { |string| } # -> self
```

Синонимы: `each_line, lines`

Перебор строк (чтение выполняется с помощью `.readlines`). **Во второй версии Ruby** синоним lines признан устаревшим.

.bytes { |byte| } *# -> self*

Синонимы: `each_byte`

Перебор байтов. **Во второй версии Ruby** синоним bytes признан устаревшим.

.chars { |char| } *# -> self*

Синонимы: `each_char`

Перебор символов. **Во второй версии Ruby** синоним chars признан устаревшим.

.codepoints { |point| } *# -> self*

Синонимы: `each_codepoint`

Перебор кодовых позиций. **Во второй версии Ruby** синоним codepoints признан устаревшим.

15.1.4. Запись данных

Поток должен быть доступен для записи.

Перед записью аргументы преобразуются в текст с помощью метода `.to_s`.

Дополнительные опции используются при открытии потока.

Опции:

encoding: внешняя кодировка. Игнорируется если указана размер фрагмента;
mode: вид создаваемого потока. Модификатор r обязателен;
perm: права доступа к файлу;
open_args: массив аргументов, используемых при открытии файла.

::write(path, string, start = nil, options = nil) *# -> string.length*

Используется для записи фрагмента текста (по умолчанию - в конец файла). После записи данных файл закрывается.

::binwrite(path, string, start = nil) *# -> string.length*

Используется для записи фрагмента в режиме ”rb:ASCII-8BIT”.

.write(object) *# -> object.to_s.bytesize*

Используется для записи текста.

.syswrite(object) *# -> object.to_s.bytesize*

Версия предыдущего метода, использующая низкоуровневые возможности системы.

.« object *# -> io*

Используется для записи текста.

.write_nonblock(string) *# -> string.length*

Используется для записи данных, не блокируя процесс выполнения, если система не готова выполнить запрос к файлу немедленно. Вместо этого программа принимает сигнал. Используется при реализации многопоточности на основе потоков выполнения или сопрограмм. Буфер программы освобождается перед выполнением записью.

.print(*object = $_) *# -> nil*

Используется для записи всех переданных аргументов.

- Если глобальная переменная `$,`, отвечающая за разделение элементов, не ссылается на nil, то она будет использоваться для разделения аргументов.
- Если глобальная переменная `$\`, отвечающая за разделение данных, не ссылается на nil, то она будет использована после записи всех объектов.

.printf(format, *objects) *# -> nil*

Используется для записи отформатированных данных `string % [*object]`.

.putc(object) *# -> object*

Используется для записи символов в поток. Переданное число считается кодовой позицией.

.puts(*object = $\ ) *# -> nil*

Используется для записи строк в поток. Аргументы разделяются с помощью символа перевода строки. Из индексного массива извлекаются все элементы.

Работа с буфером

Запись данных в файл выполняется автоматически, но не моментально. Сначала данные сохраняются в буфере программы, создаваемом интерпретатором. Поэтому бывает полезно периодически принудительно сохранять содержимое буфера на диск.

.ungetbyte(object) *# -> nil*

Используется для записи байтов в буфер (не повлияет на вызов низкоуровневых методов для чтения, например `io.sysread`).

.ungetc(char) *# -> nil*

Используется для записи символов в буфер (не повлияет на вызов низкоуровневых методов для чтения, например `io.sysread`).

.fdatasync *# -> 0*

Используется для сохранения буфера.

.fsync *# -> 0*

Используется для сохранения буфера.

.flush *# -> 0*

Используется для очищения буфера.

.sync=(bool) *# -> bool*

Используется для изменения режима синхронизации - любые записываемые данные сразу передаются операционной системе, а не записываются в буфер программы.

.sync *# -> bool*

Текущий режим синхронизации.

15.1.5. Стандартные потоки

При запуске программы автоматически создаются три стандартных потока. Они позволяют передавать данные между системой и программой. По умолчанию стандартные потоки соединены с терминалом, из которого запущена программа.

Потоки:

STDIN ($stdin) - стандартный поток для чтения (ввода). Используется для получения команд пользователя или входных данных.
STDOUT ($stdout, $>) - стандартный поток для записи (вывода). Используется для передачи данных системе;
STDERR ($stderr) - стандартный поток для записи ошибок. Используется для вывода диагностичеких и отладочных сообщений.

Отличие стандартного потока для записи от стандартного потока для записи ошибок в том что первый поток передается в конвейер, а второй нет. Стандартный поток для записи ошибок используют не только для вывода ошибок, но и для вывода любой другой информации о процессе работы программы, которая очевидно не должна попасть на вход другой программы.

Запись данных

Для записи данных в стандартный поток используются частные методы экземпляров из модуля Kernel. Они аналогичны соответствующим методам для записи в поток.

- `Kernel.print`
- `Kernel.printf`
- `Kernel.putc`
- `Kernel.puts`

.display(io = $>) *# -> nil*

Используется для записи текущего объекта в переданный поток. По умолчанию используется стандартный поток для записи.

.p(*args) *# -> args*

Используется для записи информации об объекте (`object.inspect`) в стандартный поток для записи. Несколько аргументов объединяются с использованием разделителя `$/`

.warn(message) *# -> nil*

Используется для записи предупреждений в стандартный поток для записи. Выполнение программы продолжается.

Во второй версии Ruby принимает произвольное число аргументов как `IO.puts`. Аргументы разделяются с помощью символа перевода строки. Из индексного массива извлекаются все элементы.

Чтение данных

Для чтения данных из стандартного потока, используются частные методы экземпляров `kernel.gets` и `kernel.readline`. Они аналогичны соответствующим методам для чтения строк из потока.

С помощью этих методов программа также может получать данные уже после запуска.

15.1.6. Конвейеры

Конвейер - это абстрактная сущность, объединяющая несколько процессов. Данные, записываемые в стандартный поток одним процессом, перенаправляются в стандартный поток другого процесса. В Linux создание конвейеров является главным способом взаимодействия программ запускаемых из терминала. Любой конвейер начинает работу как только появляются данные для обработки.

Обычно конвейеры анонимны и существуют только во время выполнения процесса. В отличии от них именованные конвейеры существуют в системе постоянно. Процессы периодически присоединяются к конвейеру для записи или чтения данных.

Создание анонимного конвейера: `<process> | <processs> | <process>`
Создание именованного контейнера: `mkfifo <name>`
Конвейеры можно использовать и в теле Ruby-программы.

::pipe(*encoding, options = nil) *# -> array*

```
( *encoding, options = nil ) { |read, write| } # -> object
```

Используется для создания конвейера. Если методу передан блок, то потоки отправляются в блок и закрываются после его выполнения.

В качестве аргументов принимается кодировка. Переданные опции используются при преобразовании кодировок.

```
rd, wr = IO.pipe
wr.puts "Text" # -> nil
rd.gets "Text" # -> "Text\n"
```

15.1.7. Мультиплексирование

Мультиплексирование потоков блокирует процесс выполнения, но снижает время ожидания, обрабатывая сразу несколько потоков.

::select (reads, writes = nil, errors = nil, sec = nil)

Используется для мониторинга переданных потоков.

Возвращает массив, состоящий из подмассивов потоков, готовых к работе. Обычно возвращается поток, который освободился первым. Если время ожидания истекло, а ни один поток не готов, то возвращается nil.

```
rp, wp = IO.pipe
mesg = "ping "
100.times {
  rs, ws, = IO.select([rp], [wp])
  if r = rs[0]
    ret = r.read(5)
    print ret
    case ret
    when /ping/
      mesg = "pong\n"
    when /pong/
      mesg = "ping "
    end
  end
  if w = ws[0]
    w.write(mesg)
  end
}
# ->
ping pong
ping pong
ping pong
(snipped)
ping
```

15.2. File (файлы)

Подробное описание структуры файловой системы приведено в приложении.

Файл - это фундаментальная абстракция в Linux. Linux придерживается философии ”все есть файл”, а значит большая часть взаимодействия программ реализуется через чтение и запись файлов. Операции с файлом осуществляются с помощью уникального файлового дескриптора. Основная часть системного программирования в Linux состоит в работе с файловыми дескрипторами.

Так как класс File, наследует класс IO, то с его экземплярами можно работать также, как если бы они были обычными потоками.

При открытии файла создается поток, позволяющий получить доступ к данным, сохраненным в файле.

Также в системе регистрируется файловый дескриптор - цифровой идентификатор открытого файла.

Константы:

File::SEPARATOR - символ, использующийся для разделения каталогов. В Windows обратная косая черта (\), а в Linux - косая черта (/);

File::ALT_SEPARATOR - альтернативный разделитель для каталогов;
File::PATH_SEPARATOR - символ, использующийся для разделения нескольких путей (":");
File::NULL - путь к нулевому устройству.

::new(name, mode, *perm) *# -> file*

Используется для открытия файла.

::open(name, mode, *perm) *# -> 0*

```
( name, mode, *perm )  { |file| } # -> object
```

Версия предыдущего метода, принимающая блок, после выполнения которого файл закрывается.

15.2.1. Взаимодействие с файловой системой

::delete(*path) *# -> integer*

Синонимы: `unlink`

Используется для удаления указанных файлов. Возвращается количество удаленных файлов.

::truncate(path, bytesize) *# -> 0*

Используется для уменьшения размера файла (в байтах).

.truncate(bytesize) *# -> 0*

Версия предыдущего метода, изменяющая размер текущего файла.

::rename(path, new_name) *# -> 0*

Используется для переименования файла. Невозможность операции считается исключением.

::symlink(path, name) *# -> 0*

Используется для создания символьной ссылки.

::link(path, new_name) *# -> 0*

Используется для создания жесткой ссылки. Существование файла с тем же именем считается исключением.

::utime(atime, mtime, *path) *# -> integer*

Используется для обновления времени последнего доступа и времени изменения. Возвращается количество измененных файлов.

::readlink(path) *# -> filename*

Используется для получения имени файла, связанного с символьной ссылкой.

.flock(constants) *# -> 0*

Используется для блокировки файлов. Блокировка необходима для ограничения одновременного доступа различных программ к одному файлу.

- `File::LOCK_EX` - эксклюзивная блокировки (для записи). Доступ к файлу будет запрещен до закрытия файла текущей программой. Только одна программа может устанавливать такую блокировку;

  `File::LOCK_NB` - процесс выполнения не будет ожидать окончания блокировки файла;

  `File::LOCK_SH` - совместная блокировка файла (для чтения). Блокировка применяется при чтении информации из файла несколькими программами одновременно. Произвольное число программ могут устанавливать такую блокировку;

  `File::LOCK_UN` - отмена блокировки.

15.2.2. Путь к файлу

`__FILE__` - имя выполняемого файла;
`__dir__` - путь к текущему каталогу (ruby 2.0).
Аналогично `File.dirname __FILE__`.

::path(object) *# -> path*

Используется для получения пути к объекту с помощью метода `.to_path`. Отсутствие метода считается исключением.

.path *# -> path*

Синонимы: `to_path`
Путь к файлу (переданный при его открытии).

::absolute_path(filename, basedir = File::pwd) *# -> path*

Абсолютный путь. Тильда считается частью имени каталога.

```
File.expand_path "test.rb", "~"
# -> "/home/krugloff/~/test.rb"
File.expand_path "/test.rb", ".rb" # -> "/test.rb"
```

::expand_path(filename, basedir = File::pwd) *# -> path*

Абсолютный путь. Тильда соответствует домашнему каталогу.

```
File.expand_path "test.rb", "~" # -> "/home/krugloff/test.rb"
File.expand_path "/test.rb", ".rb" # -> "/test.rb"
File.expand_path('../config/environment', "my_app/config.ru")
# -> "/home/mak/my_app/config/environment"
```

::realdirpath(filename, basedir = File::pwd) *# -> path*

Абсолютный путь. Тильда считается частью имени каталога. Отсутствие каталога считается исключением. При этом существование файла не проверяется.

```
File.realdirpath "test.rb", "~" # -> error!
File.realdirpath "/test.rb", ".rb" # -> "/test.rb"
```

::realpath(filename, basedir = File::pwd) *# -> path*

Абсолютный путь. Тильда считается частью имени каталога. Отсутствие файла считается исключением.

```
File.realpath "test.rb", "~" # -> error!
File.realpath "/test.rb", ".rb" # -> error!
```

::basename(path, extname = nil) *# -> filename*

Используется для получения имени файла из переданного пути (для разделения каталогов должна использоваться косая черта). Необязательный суффикс будет удален из результата.

```
File.basename "/test.rb", ".rb" # -> "test"
```

::dirname(path) *# -> basedir*

Используется для получения базового каталога из переданного пути (для разделения каталогов должна использоваться косая черта).

```
File.dirname "/test.rb" # -> "/"
```

::split(path) *# -> array*

Используется для разделения пути на путь к каталогу и имя файла.

```
File.split "/test.rb" # -> [ "/", "test.rb" ]
```

::extname(path) *# -> extname*

Используется для получения расширения файла. Для каталогов возвращается пустой текст.

```
File.extname "/test.rb" # -> ".rb"
```

::join(*name) *# -> path*

Используется для создания пути. Аргументы разделяются с помощью `File::SEPARATOR`.

```
File.join "/", "test", ".rb" # -> "/test/.rb"
```

::fnmatch?(pattern, path, constants = nil)

Синонимы: `fnmatch`

Проверка соответствия пути переданному образцу.

```
File.fnmatch? 'cat', 'cat'      # -> true
File.fnmatch? 'cat', 'category' # -> false
```

```
File.fnmatch? 'c{at,ub}s', 'cats'                    # -> false
File.fnmatch? 'c{at,ub}s', 'cats', File::FNM_EXTGLOB # -> true [Ruby 2.0]
```

```
File.fnmatch? 'c?t',     'cat'     # -> true
File.fnmatch? 'c??t',    'cat'     # -> false
File.fnmatch? 'c*',      'cats'    # -> true
File.fnmatch? 'c*t',     'c/a/b/t' # -> true
File.fnmatch? 'ca[a-z]', 'cat'     # -> true
File.fnmatch? 'ca[^t]',  'cat'     # -> false
```

```
File.fnmatch? 'cat', 'CAT'                     # -> false
File.fnmatch? 'cat', 'CAT', File::FNM_CASEFOLD # -> true

File.fnmatch? '?',   '/', File::FNM_PATHNAME   # -> false
File.fnmatch? '*',   '/', File::FNM_PATHNAME   # -> false
File.fnmatch? '[/]', '/', File::FNM_PATHNAME   # -> false

File.fnmatch? '\?',   '?'                       # -> true
File.fnmatch? '\a',   'a'                       # -> true
File.fnmatch? '\a',   '\a', File::FNM_NOESCAPE  # -> true
File.fnmatch? '[\?]', '?'                       # -> true

File.fnmatch? '*',   '.profile'                      # -> false
File.fnmatch? '*',   '.profile', File::FNM_DOTMATCH  # -> true
File.fnmatch? '.*',  '.profile'                      # -> true

rbfiles = '**/*.rb'
File.fnmatch? rbfiles, 'main.rb'                    # -> false
File.fnmatch? rbfiles, './main.rb'                  # -> false
File.fnmatch? rbfiles, 'lib/song.rb'                # -> true
File.fnmatch? '**.rb', 'main.rb'                    # -> true
File.fnmatch? '**.rb', './main.rb'                  # -> false
File.fnmatch? '**.rb', 'lib/song.rb'                # -> true
File.fnmatch? '*',     'dave/.profile'              # -> true

pattern = '*/*'
File.fnmatch? pattern, 'dave/.profile', File::FNM_PATHNAME  # -> false
File.fnmatch? pattern, 'dave/.profile',
  File::FNM_PATHNAME | File::FNM_DOTMATCH
# -> true

pattern = '**/foo'
File.fnmatch? pattern, 'a/b/c/foo', File::FNM_PATHNAME     # -> true
File.fnmatch? pattern, '/a/b/c/foo', File::FNM_PATHNAME    # -> true
File.fnmatch? pattern, 'c:/a/b/c/foo', File::FNM_PATHNAME  # -> true
File.fnmatch? pattern, 'a/.b/c/foo', File::FNM_PATHNAME    # -> false
File.fnmatch? pattern, 'a/.b/c/foo',
  File::FNM_PATHNAME | File::FNM_DOTMATCH
# -> true
```

::identical?(path, path2)

Сравнение двух путей.

15.2.3. Права доступа

Изменение ограничений

::chmod(*perm, *path) *# -> integer*

Используется для изменения прав доступа. Возвращается количество измененных файлов.

::lchmod(perm, *path) *# -> integer*

Версия предыдущего метода, изменяющая права доступа для символьных ссылок, а не для файлов, на которые они ссылаются.

::chown(user, group, *path) *# -> integer*

Используется для изменения владельца файла и группы владельцев. Возвращается количество измененных файлов. Только администратор может изменить владельца файла. Текущий владелец может изменить группу на любую в которую входит. Владелец и группа устанавливаются с помощью их цифровых идентификаторов. Идентификаторы 0 и -1 игнорируются.

::lchown(user, group, *path) *# -> integer*

Версия предыдущего метода, изменяющая данные для символьных ссылок, а не для файлов, на которые они ссылаются.

::umask(mask = nil) *# -> old_perm*

Используется для изменения прав доступа, присваиваемых создаваемым файлам и каталогам. Переданное число вычитается из прав доступа, используемых по умолчанию, и результат используется для объявление новых ограничений.

Предикаты

::readable?(path)

Проверка возможности чтения на основе действующих идентификаторов.

::readable_real?(path)

Проверка возможности чтения на основе реальных идентификаторов.

::world_readable?(path)

Проверка возможности чтения файла всеми пользователями (возвращаются права доступа).

::writable?(path)

Проверка возможности записи на основе действующих идентификаторов.

::writable_real?(path)

Проверка возможности записи на основе реальных идентификаторов.

::world_writable?(path)

Проверка возможности записи файла всеми пользователями (возвращаются права доступа).

::executable?(path)

Проверка возможности выполнения на основе действующих идентификаторов.

::executable_real?(path)

Проверка возможности выполнения на основе реальных идентификаторов.

::setuid?(path)

Проверка возможности выполнения файла с правами владельца.

::setgid?(path)

Проверка возможности выполнения файла с правами группы владельцев.

::sticky?(path)

Проверка наличия дополнительного свойства (sticky bit) для каталогов.

::owned?(path)

Проверка равенства идентификатора владельца файла и действующего идентификатора (текущий пользователь является владельцем файла).

::grpowned?(path)

Проверка равенства цифрового идентификатора текущей группы и цифрового идентификатора группы владельцев файла (текущий пользователь относится к владельцам файла).

15.2.4. Статистика

::stat(path) *# -> a_file_stat*

Используется для получения экземпляра класса File::Stat, содержащего информацию о файле.

::lstat(path) *# -> a_file_stat*

Версия предыдущего метода, сохраняющая данные для символьных ссылок, а не для файлов, на которые они ссылаются.

.lstat *# -> file_stat*

Версия предыдущего метода, возвращающая информацию о текущем файле.

::atime(path) *# -> time*

Время последнего доступа к файлу.

.atime *# -> time*

Время последнего доступа к текущему файлу.

::ctime(path) *# -> time*

Время последнего изменения информации о файле (но не самого файла).

.ctime *# -> time*

Время последнего изменения информации о текущем файле (но не самого файла).

::mtime(path) *# -> time*

Время последнего изменения файла.

.mtime *# -> time*

Время последнего изменения текущего файла.

::size(path) *# -> integer*

Размер файла.

.size *# -> integer*

Размер текущего файла.

::ftype(path) *# -> string*

Тип файла.

- *"file"* - обычный файл;
  "directory" - каталог;
  "blockSpecial" - блочное устройство;
  "characterSpecial" - символьное устройство;
  "fifo" - конвейер;
  "link" - ссылка;
  "socket" - сокет;
  "unknown" - неизвестный тип.

15.2.5. Предикаты

::exist?(path)

Синонимы: `exists?`

Проверка существования файла.

::size?(path) *# -> integer*

Проверка существования файла. Для существующих файлов возвращается их размер.

::zero?(path)

Проверка существования файла нулевого размера.

::file?(path)

Проверка существования обычного файла.

::symlink?(path)

Проверка существования символьной ссылки.

::directory?(path)

Проверка существования каталога.

::blockdev?(path)

Проверка существования блочного устройства.

::chardev?(path)

Проверка существования символьного устройства.

::pipe?(path)

Проверка существования конвейера.

::socket?(path)

Проверка существования сокета.

15.3. Dir (каталоги)

Добавленные модули: Enumerable

Каталог - это файл, содержащий информацию о расположении какой-либо группы файлов.

- *Базовый каталог* - каталог, в котором находится файл;
  Рабочий каталог - каталог, относительно которого выполняется программа. Поиск файлов выполняется относительно рабочего каталога;
  Домашний каталог - персональный каталог пользователя, зарегистрированного в системе.

::new(path) *# -> dir*

Используется для создания нового объекта.

::open(path) *# -> dir*

```
(path) { |dir| } # -> dir
```

Версия предыдущего метода, позволяющая передавать объект в блок. После выполнения блока каталог закрывается.

.close *# -> nil*

Используется для закрытия каталога. Любая попытка использования объекта считается исключением.

15.3.1. Работа с файловой системой

::mkdir(name, *perm) *# -> 0*

Используется для создания каталога. Невозможность выполнения операции считается исключением.

::delete(path) *# -> 0*

Синонимы: `rmdir, unlink`

Ипсользуется для удаления каталога. Удаление каталога, содержащего файлы, считается исключением.

::chdir(path = Dir.home) *# -> 0*

```
( path = Dir.home ) { |path| } # -> object
```

Используется для изменения рабочего каталога либо для всей программы, либо только во время выполнения блока.

::chroot(path) *# -> 0*

Используется для изменения корневого каталога в Unix-подобных системах. Программа, запущенная с изменённым корневым каталогом, будет иметь доступ только к файлам, содержащимся в данном каталоге. Только пользователь с правами администратора может изменить корневой каталог программы.

15.3.2. Содержимое каталога

::entries(path) *# -> array*

Имена всех файлов, содержащихся в каталоге.

::glob(pattern, constants = nil) *# -> array*

```
( pattern, constants = nil ) { |name| } # -> nil
```

Используется для поиска файлов (синтаксис шаблонов описан в приложении). Чувствительность к регистру зависит от ОС. Найденные совпадения могут передаваться в блок. Дополнительно могут быть переданы константы, влияющие на поиск.

::[pattern] *# -> array*

Аналогично выполнению `Dir.glob pattern, 0`.

.read *# -> name*

Имя следующего файла. При достижении конца каталога возвращается nil.

.pos *# -> integer* Синонимы: `tell`

Текущая позиция в каталоге.

.seek(pos) *# -> self*

Используется для изменения позиции поиска.

.pos=(pos) *# -> integer*

Используется для изменения позиции поиска.

.rewind *# -> self*

Используется для сброса текущей позиции поиска.

15.3.3. Итераторы

.each { |name| } *# -> self*

Перебор имен файлов.

::foreach(path) { |name| } *# -> nil*

Перебор имен файлов.

15.3.4. Остальное

::directory?(path)

Синонимы: `exist?, exists?`

Проверка относится ли файл к каталогам или к ярлыкам, ссылающимся на каталог.

::getwd *# -> path*

Синонимы: `pwd`

Путь к рабочему каталогу программы.

::home(user = nil) *# -> path*

Путь к домашнему каталогу текущего пользователя или пользователя с переданным идентификатором.

.inspect *# -> string*

Информация об объекте.

.path *# -> path*

Путь к каталогу (переданный при создании объекта).

15.4. Информация о файле

.test(type, first_path, second_path = nil)

Информация о переданных файлах.

Тип информации:

?A - время последнего доступа к файлу;
?C - время последнего изменения информации о файле;
?M - время последнего изменения файла;
?e - проверка существования файла;
?s - размер файла. Для фалов нулевого размера возвращается nil;
?z - проверка существования файла нулевого размера;
?f - проверка существования обычного файла;
?L - проверка существования символьной ссылки;
?d - проверка существования каталога;
?b - проверка существования блочного устройства;
?c - проверка существования символьного устройства;
?S - проверка существования сокета;
?p - проверка существования конвейера;
?r - проверка возможности чтения на основе действующих идентификаторов;
?R - проверка возможности чтения на основе реальных идентификаторов;
?w - проверка возможности записи на основе действующих идентификаторов;
?W - проверка возможности записи на основе реальных идентификаторов;
?x - проверка возможности выполнения на основе действующих идентификаторов;
?X - проверка возможности выполнения на основе реальных идентификаторов;
?u - проверка возможности выполнения файла с правами владельца;
?g - проверка возможности выполнения файла с правами группы владельцев;
?K - проверка существования дополнительного свойства (sticky bit) для каталогов;
?o - проверка равенства идентификатора владельца файла и действующего идентификатора (текущий пользователь является владельцем файла);
?O - проверка равенства идентификатора владельца файла и реального идентификатора (текущий пользователь является владельцем файла);
?G - проверка равенства цифрового идентификатора текущей группы и цифрового идентификатора группы владельцев файла (текущий пользователь относится к владельцам файла);
?- - проверка ссылаются ли два пути на один и тот же файл;
?= - проверка равенства времени последнего изменения двух файлов;
?< - используется для сравнения времени последнего изменения двух файлов;
?> - используется для сравнения времени последнего изменения двух файлов.

15.4.1. Класс File::Stat

Экземпляры класса получаются с помощью методов `file.stat` и `file.lstat`. Это позволяет получать полную информацию о файле, обратившись к системе только один раз.

::new(path) *# -> stat*

Используется для сбора информации о файле. Отсутствие файла считается исключением.

Приведение типов

.inspect *# -> string*

Информация об объекте.

Операторы

.<=>(stat)

Используется для сравнения времени сбора информации.

Права доступа

Для проверки прав доступа определены методы, аналогичные методам экземпляров класса File.

- `.readable?`
  `.readable_real?`
  `.world_readable?`
  `.writable?`
  `.writable_real?`
  `.world_writable?`
  `.executable?`
  `.executable_real?`
  `.setuid?`
  `.setgid?`
  `.sticky?`
  `.owned?`
  `.grpowned?`

.uid *# -> integer*

Цифровой идентификатор владельца файла.

.gid *# -> integer*

Цифровой идентификатор группы владельцев файла.

.mode *# -> perm*

Права доступа.

Статистика

Для получения прав доступа определены методы, аналогичные методам из класса File.

- `.atime`
  `.ctime`
  `.mtime`
  `.size`
  `.ftype`

.blksize *# -> integer*

Системный размер блока, выделенный для файла. Для систем, не поддерживающих эту информацию возвращается nil.

.blocks *# -> integer*

Системный номер блока, выделенный для файла. Для систем, не поддерживающих эту информацию возвращается nil.

.dev *# -> integer*

Цифровой код устройства, на котором размещена информация о файле.

.dev_major *# -> integer*

Основная часть цифрового кода устройства.

.dev_minor *# -> integer*

Дополнительная часть цифрового кода устройства.

.rdev *# -> integer*

Цифровой код устройства, на котором размещена информация о файле.

.rdev_major *# -> integer*

Основная часть цифрового кода устройства.

.rdev_minor *# -> integer*

Дополнительная часть цифрового кода устройства.

.ino *# -> integer*

Inode код файла. С помощью этого кода получается информация о файле от системы.

.nlink *# -> integer*

Цифровой код жесткой ссылки.

Предикаты

Определены методы, аналогичные методам из класса File.

- `.zero?`
  `.file?`
  `.symlink?`
  `.directory?`
  `.blockdev?`
  `.chardev?`
  `.pipe?`
  `.socket?`

15.4.2. Модуль FileTest

Права доступа

Для проверки прав доступа определены методы, аналогичные методам из класса File.

- `FileTest::readable?(path)`
  `FileTest::readable_real?(path)`
  `FileTest::world_readable?(path)`
  `FileTest::writable?(path)`
  `FileTest::writable_real?(path)`
  `FileTest::world_writable?(path)`
  `FileTest::executable?(path)`
  `FileTest::executable_real?(path)`
  `FileTest::setuid?(path)`
  `FileTest::setgid?(path)`
  `FileTest::sticky?(path)`
  `FileTest::owned?(path)`
  `FileTest::grpowned?(path)`

Предикаты

Определены методы, аналогичные методам из класса File.

- `FileTest::exist?(path)` (Синонимы: `exists?`)
  `FileTest::size?(path) # -> integer`
  `FileTest::zero?(path)`
  `FileTest::file?(path)`
  `FileTest::symlink?(path)`
  `FileTest::directory?(path)`
  `FileTest::blockdev?(path)`
  `FileTest::chardev?(path)`
  `FileTest::pipe?(path)`
  `FileTest::socket?(path)`

Остальное

Определены методы, аналогичные методам из класса File.

- `FileTest::size(path) # -> integer`
  `FileTest::identical?( first_path, second_path )`

Глава 16

Обработка аргументов

Аргументы, переданные при запуске программы, сохраняются в массиве ARGV.

16.1. Файлы

Программы, работающие с файлами, могут принимать как по одному файлу, так и несколько сразу. Для работы с одним файлом используется массив ARGV, а для работы с несколькими файлами - поток ARGF.

16.1.1. ARGV

Для чтения файла, переданного при запуске программы, используются частные методы экземпляров `kernel.gets` и `kernel.readline`. Они аналогичны соответствующим методам для чтения строк из потока.

16.1.2. ARGF

Добавленные модули: Enumerable

ARGF ($<) - это поток, открываемый для всех файлов, содержащихся в ARGV. При этом подразумевается, что ARGV содержит только пути к файлам.

Файлы обрабатываются в том порядке, в котором они содержатся в ARGV (порядок, в котором они были переданы при запуске). После обработки путь к файлу удаляется автоматически.

Файлы обрабатываются последовательно в виде одного виртуального файла. Концом потока считается конец последнего файла, а не конец отдельных элементов.

Если ARGV ссылается на пустой массив, то ARGF ссылается на стандартный поток для чтения.

Управление потоком

Для управления потоком используются методы класса. Они аналогичны соответствующим методам для управления потоком.

- `ARGF::binmode`
- `ARGF::close` (обработка всех файлов считается исключением `IOError`)
- `ARGF::skip` (при отсутствии файлов ничего не выполняется)
- `ARGF::binmode?`

- `ARGF::closed?`
- `ARGF::eof?` (Синонимы: `eof`)

::argv *# -> ARGV*

::filename *# -> path*

Синонимы: `path`

Относительный путь к обрабатываемому файлу. При взаимодействии с стандартным потоком для ввода возвращается `"-"`. Аналогично использованию $FILENAME.

Кодировка

Для изменения кодировки используются методы класса. Они аналогичны соответствующим методам для изменения кодировки потока.

- `ARGF::external_encoding`
- `ARGF::internal_encoding`
- `ARGF::set_encoding`

Приведение типов

::to_s *# -> "ARGF"*

::to_io *# -> io*

Синонимы: `file`

Используется для получения потока для обрабатываемого файла (или стандартного потока для ввода).

::to_i *# -> integer*

Синонимы: `fileno`

Используется для получения дескриптора обрабатываемого файла. Отсутствие файлов считается исключением `ArgumentError`.

::to_a(sep = $/, size = nil) *# -> array*

Синонимы: `readlines`

Используется для извлечения всех строк и сохранения их в индексном массиве (размер массива может быть ограничен). Переданный разделитель обрабатывается в качестве символа перевода строки (если передается пустой текст, то обрабатывается "\n\n").

::to_write_io *# -> io*

Используется для получения потока, доступного для записи (только если используется режим редактирования файлов - при запуске программы передан ключ `-i`).

Чтение данных

Для чтения содержимого используются методы класса. Они аналогичны соответствующим методам для чтения содержимого потока.

Фрагменты:

- `ARGF::read`
- `ARGF::read_nonblock`
- `ARGF::readpartial`

Строки:

- + `ARGF::gets`
- + `ARGF::readline`
- + `ARGF::lineno`
- + `ARGF::lineno=(pos)`
- + `ARGF::seek`
- + `ARGF::rewind`

Символы:

- `ARGF::getc`
- `ARGF::readchar`

Байты:

- `ARGF::getbyte`
- `ARGF::readbyte`
- `ARGF::pos` (Синонимы: `tell`)
- `ARGF::pos=(pos)`

Итераторы:

- `ARGF::each` (Синонимы: `each_line, lines`)
- `ARGF::bytes` (Синонимы: `each_byte`)
- `ARGF::chars` (Синонимы: `each_char`)
- `ARGF::codepoints` (Синонимы: `each_codepoint`, Ruby 2.0)

Запись данных

Запись данных с помощью ARGF возможна только при запуске программы с ключом `-i`. Запись данных выполняется относительно текущего обрабатываемого файла. Для записи используются методы класса. Они аналогичны соответствующим методам для записи данных в поток.

::inplace_mode *# -> string*

Расширение, применяемое при создании резервных копий изменяемых файлов.

::inplace_mode=(ext) *# -> self*

Используется для изменения расширения, применяемого при создании резервных копий изменяемых файлов.

- `ARGF::write`
- `ARGF::print`
- `ARGF::printf`
- `ARGF::putc`
- `ARGF::puts`

Глава 17

Библиотеки кода

Чтобы облегчить повторное использование кода, программу принято разделять на файлы, которые могут быть выделены в отдельные библиотеки. Библиотека - это группа файлов, содержащих набор модулей и классов (обычно каждый файл содержит определение одного модуля или класса).

17.1. Использование

Для использования библиотеки требуется явно объявить это интерпретатору. В свою очередь, интерпретатор автоматически находит и извлекает содержимое библиотеки. Обычно используемые библиотеки объявляются в начале программы.

Поиск всех объявленных библиотек происходит в каталогах, хранящихся в глобальном массиве `$LOAD_PATH ($:)` (изменение значения элементов запрещено в Ruby 2.0; для объектов не относящихся к String вызывается метод `object.to_path`). Поиск файла выполняется, начиная с первого элемента (начиная с первого каталога).

.require(path) *# -> bool*

Используется для однократной загрузки библиотеки. Названия загруженных библиотек сохраняются в массиве `$LOAD_FEAUTURES ($")` (изменение значения элементов запрещено в Ruby 2.0; для объектов не относящихся к String вызывается метод `object.to_path`). Каждая библиотека может быть загружена только один раз. Уровень безопасности объявляемой библиотеки должен быть равен 0.

На самом деле любую библиотеку можно загружать произвольное число раз, если перед этим удалять её название из массива.

В названии файла библиотеки расширение обычно не указывается. По умолчанию обрабатывается расширение `.rb`. Если файла с таким расширением не найдено, то будет произведен поиск бинарного файла с тем же именем (например, с расширениями `.so` или `.dll`).

.require_relative(path) *# -> bool*

Версия предыдущего метода, использующаяся для поиска библиотек относительно базового каталога программы.

.load(path, anonym = false)

Используется для многократной загрузки библиотеки. В имени файла должно быть указано его расширение.

Код библиотеки может быть выполнен в теле анонимного модуля. В этом случае она не будет влиять на глобальную область видимости основной программы.

.autoload(name, path) *# -> nil*

Используется для автоматизации загрузки библиотек (отложенная загрузка). Поиск библиотеки выполняется только при вызове соответствующей константы.

.autoload?(name) *# -> path*

Используется для получения названия библиотеки, загружаемой при получении соответствующей константы. Если такая библиотека не объявлена, то возвращается nil.

Для загрузки библиотек относительно констант вызываемых в теле определенного модуля существуют версии методов `module.autoload` и `module.autoload?`.

17.2. Усовершенствование (Ruby 2.0)

Во второй версии Ruby добавлен механизм, позволяющий усовершенствовать используемые библиотеки кода, с помощью создания улучшений. Улучшение - это синтаксическая конструкция, позволяющая переопределять существующее поведение. В отличии от обычного переопределения, улучшения воздействуют только на текущую область видимости.

Улучшения считаются экспериментальной функцией и их применение для рабочих приложений не рекомендуется.

Достоинства:

- Применение изменений только в текущей области видимости.

Недостатки:

- Усложнение понимания кода.
- Усложнение поиска методов.
- Результат выполнения кода зависит от его местоположения.

.refine(a_class) { } *# -> a_module [private Module]*

Используется для улучшения переданного класса. Метод создает анонимный модуль, содержащий сделанные улучшения (self ссылается на этот модуль). Модули могут содержать сразу несколько улучшений. Метод существует только в теле модуля (но не класса).

Улучшения действуют только на сущности, создаваемые после применения улучшения.

```
# Старый способ
class String
  def bang
    "#{self}"
  end
end
"Hello".bang # -> "Hello"
```

```
# Новый способ
module StringBang
  "Hello".bang # -> NoMethodError!

  refine String do
    def bang; "#{self}"; end
  end

  "Hello".bang # -> "Hello!"
end
"Hello".bang # -> NoMethodError!
```

.using(module) *# -> main [MAIN]*

Используется для применения улучшений из модуля. Улучшения могут применяться только для файла и в методах `Kernel.eval`, `Kernel.instance_eval` или `Kernel.module_eval`.

- Улучшения действуют только на сущности, создаваемые после применения. При наличии улучшений класса, поиск методов начинается с улучшений (улучшения добавляются в начало очереди вызова и могут быть доступны с помощью super).

```
module Foo
  def foo
    puts "C#foo in Foo"
  end
end

class C
  prepend Foo
  def foo
    puts "C#foo"
  end
end

class D < C
  def foo
    super
  end
end
```

```
module M
  refine C do
    def foo
      puts "C#foo in M"
    end
  end
end

C.new.foo # -> 'C#foo in Foo'

using M
C.new.foo # -> "C#foo in M"
D.new.foo # -> "C#foo in Foo"

class E < C
  def foo
    super
  end
end

E.new.foo # -> "C#foo in M"
```

- Улучшения не действуют на методы, определяемые вне улучшаемого контекста. Улучшения могут не действовать во время вызова `Kernel.send`, `Kernel.method`, и `Kernel.respond_to?`.

```
C = Class.new

module M
  refine C do
    def foo
      puts "C#foo in M"
    end
  end
end

def call_foo(x)
  x.foo
end

using M

x = C.new
x.foo       # -> "C#foo in M"
x.send :foo # -> NoMethodError!
x.respond_to? :foo # -> false
call_foo(x) # -> NoMethodError!
```

- Улучшения действую на методы, которые были определены после применения улучшений, даже если метод вызывается вне действия улучшения.

```
# c.rb:

class C
end
```

```
# m.rb:

require "c"

module M
  refine C do
    def foo
      puts "C#foo in M"
    end
  end
end
```

```
# m_user.rb:

require "m"

using M

class MUser
  def call_foo(x)
    x.foo
  end
end
```

```
# main.rb:

require "m_user"

x = C.new
m_user = MUser.new
m_user.call_foo(x)  # -> C#foo in M
x.foo               # -> NoMethodError!
```

- Улучшения не действуют, если метод `.using` не вызывался.

```
# В файле:

# not activated here
using M
# activated here
class Foo
  # activated here
  def foo
    # activated here
  end
  # activated here
end
# activated here
```

```
# В eval:

# not activated here
eval <<EOF
  # not activated here
  using M
  # activated here
EOF
# not activated here
```

```
# В условии:
# not activated here
if false
  using M
end
# not activated here
```

- Переопределение метода, добавленного улучшением, может привести к аварийному останову программы.

Глава 18

Исключения

Исключение - это допустимая ошибка, возникающая в процессе выполнения программы, прерывающая выполнение до тех пор, пока не ислючение не будет обработано. Если исключение не обрабатывается, то выполнение программы прекращается.

Каждое исключение относится к классу Exception и его производным. Обычно название каждого подкласса, содержит полную информацию о причине возникновения исключения.

Для создания новых типов исключений обычно используются классы StandartError и RuntimeError.

Системные ошибки, имеющие стандартный цифровой код, также относятся к исключениям. Модуль Errno динамически связывает полученные от операционной системы цифровые коды с подклассами Exception. При этом для каждой ошибки создается собственный подкласс `SystemCallError`, на который ссылается константа в модуле Errno. Цифровой код ошибки может быть получен с помощью константы Errno (`Errno::<ErrorKlass>::Errno`).

18.1. Иерархия исключений

- *Exception* - базовый класс для всех исключений.
  - *NoMemoryError* - выделение памяти не может быть выполнено;
    ScriptError- базовый класс для ошибок интерпретации;
    * *LoadError* - файл не может быть загружен;
      NotImplemenetedError - метод не поддерживается системой;
      SyntaxError - ошибка в синтаксисе;

    SecuirityError - нарушение требований безопасности;
    SignalException - получение сигнала от системы;
    * *Interrupt* - сигнал прервать процесс выполнения (обычно Ctrl+C);

    SystemExit - завершение выполнения программы системой;
    SystemStackError - переполнение стека;
    StandardError - базовый класс для стандартных ошибок выполнения;
    * *Math::DomainError* - объекты не принадлежат области определения функции;
      ArgumentError - ошибка при передаче аргументов;
      EncodingError - базовый класс для ошибок, связанных с кодировкой;

- *Encoding::CompatibilityError* - исходная кодировка не совместима с требуемой;
  Encoding::ConverterNotFoundError - требуемая кодировка не поддерживается;
  Encoding::InvalidByteSequenceError - текст содержит некорректные байты;
  Encoding::UndefinedConversionError - текст содержит неопределенные символы;

FiberError - ошибка при работе с управляемыми блоками;
IOError - возникновение ошибки при работе с потоками;

- *EOFError* - достигнут конец файла;

IndexError - индекс не найден;

- *KeyError* - ключ не найден;
  StopIteration - завершение итерации;

LocalJumpError - блок не может быть выполнен;
NameError - неизвестный идентификатор;

- *NoMethodError* - неизвестный метод;

RangeError - выход за границы диапазона;

- *FloatDomainError* - попытка преобразования констант для определения специальных чисел (NaN и т.д.);

RegexpError - ошибка в регулярном выражении;
RuntimeError - универсальный класс для ошибок выполнения;
SystemCallError - базовый класс для системных ошибок;
ThreadError - ошибка при работе с процессами;
TypeError - неправильный тип объекта. Данное исключение также возникает при объявлении наследования для существующего класса;
ZeroDivisionError - деление целого числа на ноль.

18.2. Методы

18.2.1. Exception

::exception(message = nil) *# -> exception*

Используется для создания объекта. Для аргумента вызывается метод `object.to_str`.

::new(mesage = nil) *# -> exception*

Используется для создания объекта.

.exception(message = nil) *# -> exception*

Используется для получения нового экземпляра того же класс. Для аргумента вызывается метод `.to_str`.

.backtrace *# -> array*

Используется для получения данных о распространении исключения. Каждый элемент имеет вид:

```
"имя_файла:номер_строки: in 'идентификатор_метода'"
```

или

```
"имя_файла:номер_строки"
```

.set_backtrace(array) *# -> array*

Используется для изменения данных о распространении исключения.

.to_s *# -> string*

Синонимы: `message`

Сообщение об ошибке (или идентификатор класса).

.inspect *# -> string* Идентификатор класса.

18.2.2. LoadError [ruby 2.0]

.path *# -> string*

Метод используется для получения пути, по которому не был найден файл.

18.2.3. SignalException

::new(sig_name) *# -> a_signal_exception*

```
(sig_number, name = nil)
```

Метод используется для создания нового объекта. Название сигнала должно быть известно интерпретатору.

.signo *# -> number*

Метод используется для получения номера сигнала.

18.2.4. SystemExit

::new(status = 0) *# -> exception*

Используется для создания нового объекта.

.status *# -> integer* Статус завершения программы.

.success? *# -> bool*

Проверка удалось ли завершение программы.

18.2.5. Encoding::InvalidByteSequenceError

.destination_encoding *# -> encoding* Требуемая кодировка

.destination_encoding_name *# -> string* Название требуемой кодировки.

.source_encoding *# -> encoding*

Исходная кодировка. При нескольких преобразованиях исходной будет считаться последняя стабильная кодировка.

.source_encoding_name *# -> string*

Название исходной кодировки. При нескольких преобразованиях исходной будет считаться последняя стабильная кодировка.

.error_bytes *# -> string*

Байт из-за которого возникло исключение.

.incomplete_input? *# -> bool*

Проверка возникновения исключения из-за преждевременного завершения текста.

.readagain_bytes *# -> string*

Байт, обрабатываемый в момент возникновения исключения.

18.2.6. Encoding::UndefinedConversionError

.destination_encoding *# -> encoding* Требуемая кодировка

.destination_encoding_name *# -> string* Название требуемой кодировки.

.source_encoding *# -> encoding*

Исходная кодировка. При нескольких преобразованиях исходной будет считаться последняя стабильная кодировка.

.source_encoding_name *# -> string*

Название исходной кодировки. При нескольких преобразованиях исходной будет считаться последняя стабильная кодировка.

.error_char *# -> string*

Символ из-за которого возникла ошибка.

18.2.7. StopIteration

.result *# -> object* Результат итерации.

18.2.8. LocalJumpError

.exit_value *# -> object*

Аргумент, передача которого привела к возникновению исключения.

.reason *# -> symbol*

Идентификатор инструкции, выполнение которой привело к возникновению исключения (:break, :redo, :retry, :next, :return, или :noreason).

18.2.9. NameError

::new(message, name = nil) *# -> exception*

Используется для создания нового объекта.

.name *# -> name*

Идентификатор, использование которого привело к возникновению исключения.

18.2.10. NoMethodError

::new(message, name, *args) *# -> exception*

Используется для создания нового объекта.

.args *# -> object*

Аргументы, переданные отсутствующему методу.

18.2.11. SystemCallError

::new(message, integer) *# -> exception*

Используется для создания нового экземпляра класса из модуля Errno (если методу передан известный системе цифровой код ошибки) или класса SystemCallError.

.errno *# -> integer* Цифровой код ошибки.

18.3. Возникновение и обработка исключений

18.3.1. Вызов исключения

Вызов исключения выполняется с помощью частного метода экземпляров из модуля Kernel.

.raise(message = nil) *# -> exception*

```
( exc = RuntimeError, message = nil, pos = caller ) # -> exception
```

Синонимы: `fail`

Используется для повторного вызова последнего исключения или создания нового (RuntimeError), если `$!` ссылается на nil.

В другом случае методу передаются любой объект, отвечающий на вызов метода `.exception`, сообщение об ошибке и текущая позиция выполнения программы.

18.3.2. Обработка исключений

Обработка событий выполняется с помощью предложения rescue, которое может использоваться только в теле предложений begin, def, class, или module.

Исключения обрабатываются в том же порядке, в котором объявляются обработчики. При возникновении исключения интерпретатор останавливает процесс выполнения программы и начинает поиск обработчика, продвигаясь вверх по области вызова.

Если исключения возникло в результате обработки другого исключения, то поиск обработчиков выполняется заново.

После обработки исключения выполнение программы не продолжается.

Полный синтаксис

```
begin
  тело_предложения
rescue
  тело_обработчика
else
  code
ensure
  code
end
```

- Тело обработчика выполняется после возникновения исключения в теле предложения. Переменная `$!` при этом ссылается на конкретный экземпляр исключения.

  Чтобы инициализировать локальную переменную используют инструкцию `rescue => локальная_переменная`.

- По умолчанию обрабатываются экземпляры StandardError и его производных.

  Для ограничения типов обрабатываемых исключений используют инструкцию `rescue class` или `rescue class => локальная_переменная`. Несколько классов разделяются запятыми.

- Инструкция else выполняется если исключений не получено. При этом исключения, возникшие в теле инструкции не обрабатываются.

- Инструкция ensure выполняется после выполнения всего предложения. Результат ее выполнения не влияет на результат выполнения предложения (кроме случаев использования инструкций return, break и т.д)

Краткий синтаксис:

```
код rescue тело_обработчика
```

Если в коде будет вызвана ошибка, то выполняется тело обработчика. Обрабатываются только экземпляры StandardError и его производных.

18.3.3. Catch и Throw

В других языках программирования обработка событий обычно выполняется с помощью пары инструкций catch и throw. В Ruby существуют частные методы экземпляров из модуля Kernel, ведущие себя сходным образом.

.catch(name = nil) { |name| } *# -> object*

Используется для создания прерываемого фрагмента кода. Выполнение останавливается при вызове метода `object.throw` с тем же идентификатором. При вызове без аргументов новый случайный идентификатор передается блоку.

.throw(name, *args)

Используется для завершения выполнения блока, переданного методу `object.catch` с тем же идентификатором (иначе возникает исключение). Поиск блока выполняется вверх по иерархии области видимости. Дополнительные аргументы возвращаются методом `object.catch`.

Глава 19

Тестирование и отладка

19.1. Тестирование

Если вы хотите улучшить программу, вы должны не тестировать больше, а программировать лучше.

19.1.1. Основы

Любая, даже самая маленькая программа, не может гарантировать правильной работы. Тестирование программ и исправление обнаруженных ошибок - один из самых трудоемких этапов разработки. Он не заканчивается и после публикации приложения.

Тестирование - это средство обнаружения ошибок. Для поиска и устранения их причин выполняется отладка. Устранение дефектов - это дорогой и длительный процесс. Легче сразу создать высококачественную программу, чем создать низкокачественную и исправлять ее.

Тесты создаются для обнаружения ошибок, которые никогда не должны происходить, в отличии от обработки исключений, возникновение которых возможно и предусмотрено реализацией.

Обзор кода часто эффективнее, чем тестирование, но выполнение тестов позволяет проверить влияние изменений, внесенных в ходе разработки.

Существует множество подходов к тестированию приложения, но в основном грамотное тестирование - процесс прежде всего творческий.

В общем случае тестирование приложения разделяется на четыре уровня:

- *Модульное тестирование* - тестирование минимально возможного фрагмента кода (unit test);
  Интеграционное тестирование - тестирование взаимодействия между различными элементами приложения;
  Функциональное тестирование - тестирование задач, выполняемых с помощью приложения;
  Тестирование производительности - тестирование скорости выполнения программы и затрачиваемых на это ресурсов компьютера.

Обычно для облегчения тестирования код разделяют на работающий с проверенными и непроверенными данными.

При выполнении теста ему передаются как заведомо правильные, так и заведомо неправильные данные.

Для тестирования программы может быть использована команда `testrb` (исполняемый файл на Ruby, использующий стандартную библиотеку Test::Unit).

Программа принимает путь к файлу с тестами (или выполняет все тесты в каталоге) и выводит результатаы тестирования.

Для проверки выполнения небольших кусков кода может быть использован интерактивный терминал irb.

19.1.2. TDD

Одна из популярных техник тестирования - разработка, управляемая тестами (TDD, Test-Drive Development). Использование этой техники разделено на следующие этапы:

- Написание кода, тестирующего часть приложения;
- Выполнение теста. Получение отрицательного результата (этот этап необходим для проверки корректности теста);
- Написание кода приложения;
- Выполнение теста. Получение положительного результата.

Создание тестов перед кодом фокусирует внимание на требованиях к программе (т.е. необходимо понимание для чего она создается).

Тестирование -> написание кода -> удовлетворение требований -> улучшение кода (рефакторинг).

19.2. Отладка

19.2.1. Состояние программы

Так как вся программа по сути является объектом, то интроспекция также позволяет узнать о состоянии выполнения программы.

.global_variables *# -> array [PRIVATE: Kernel]*

Идентификаторы глобальных переменных.

.local_variables *# -> array [PRIVATE: Kernel]*

Идентификаторы локальных переменных.

::constants *# -> array [Module]*

Идентификаторы всех констант в теле программы (в теле Object).

::nesting *# -> array [Module]*

Очередь вызовов метода.

.__method__ *# -> symbol [Kernel]*

Синонимы: `__callee__`

Идентификатор текущего метода. Вне тела метода возвращается nil.

Во второй версии Ruby возвращается оригинальное название метода, а не имя синонима.

```
def first; __callee__; end
first # -> :first

alias second first
second # -> :first
```

Переменные и константы

RUBY_PATCHLEVEL - версия интерпретатора;
RUBY_PLATFORM - название используемой системы;
RUBY_RELEASE_DATE - дата выпуска интерпретатора;
RUBY_VERSION - версия языка;
$PROGRAM_NAME ($0) - имя выполняемой программы (по умолчанию - имя файла с расширением);
__dir__ - путь к текущему каталогу (ruby 2.0).
Аналогично `File.dirname __FILE__`;
__FILE__ - имя выполняемого файла;
__LINE__ - номер выполняемой строки кода;
__Encoding__ - кодировка программы.

19.2.2. Стек выполнения

.caller(offset = 1) *# -> array [Kernel]*

Состояние стека выполнения в виде массива, содержащего: `"файл:строка_кода"` или `"файл: строка_кода in метод"`.

Во второй версии Ruby второй аргумент влияет на размер результата. Если он больше, чем количество выполненных строк кода, то возвращается nil.

Ruby 2.0

Во второй версии добавлен новый способ получить доступ к состоянию стека выполнения программы.

.caller_locations(start = 1, length = nil) *# -> array or nil [Ruby 2.0]*

```
(range) # -> array or nil
```

Фрагмент состояния стека выполнения программы в виде массива, содержащего экземпляры `Thread::Backtrace::Location`.

Если начальная позиции фрагмента превышает текущий размер стека, то возвращается nil.

```
# Ruby 1.9:
def whoze_there_using_caller
  caller[0][/`([^']*)'/, 1]
end

# Ruby 2.0:
def whoze_there_using_locations
  caller_locations(1, 1)[0].label
end
```

Thread::Backtrace::Location

.absolute_path *# -> string*

Полный путь к файлу.

```
caller_locations.last.absolute_path # -> "/usr/bin/irb"
```

.base_label *# -> string*

Основная метка. Обычно соответсвует простой метке без дополнительного оформления.

```
caller_locations.last.base_label # -> "main"
```

.inspect *# -> string*

Информация об объекте.

```
caller_locations.last.inspect # -> "\"/usr/bin/irb:12:in '<main>'\""
```

.label *# -> string*

Метка. Обычно содержит название метода, класса, модуля и т.д. с дополнительным оформлением.

```
caller_locations.last.label # -> "<main>"
```

.lineno *# -> integer*

Номер строки кода.

```
caller_locations.last.lineno # -> 12
```

.path *# -> string*

Имя файла.

```
loc = caller_locations(0..1).first
loc.path # -> 'caller_locations.rb'
```

.to_s *# -> string*

Иноформация об объекте в стиле метода `Kernel.caller`.

```
caller_locations.last.to_s # -> "/usr/bin/irb:12:in '<main>'"
```

19.2.3. Трассировка

Трассировка программы может выполняться с помощью частных методов экземпляров из модуля Kernel.

.set_trace_func(proc = nil)

Используется для выполнения переданной подпрограммы при возникновении ряда событий. Трассировка в теле подпрограммы при этом не выполняется.

Подпрограмме передаются: идентификатор события, имя файла, номер строки кода, цифровой идентификатор объекта, экземпляр класса Binding и идентификатор класса объекта.

Передача nil отменяет трассировку.

Возможные события:

- *"c-call"* - вызов Си функции;
  "c-return" - завершение выполнения Си функции;
  "call" - вызов Ruby метода;
  "return" - завершение выполнения Ruby метода;
  "class" - начало определения класса или модуля;
  "end" - завершение определения класса или модуля;
  "line" - выполнение новой строки кода;
  "raise" - вызов ошибки.

Метод признан устаревшим во второй версии Ruby. Вместо него используется класс TracePoint.

.trace_var(name, code) *# -> nil*

```
(name) { |object| } -> nil
```

Используется для выполнения кода при изменении глобальных переменных. В блок передается новое значение.

.untrace_var(name, code = nil) *# -> array*

Используется для отмены трассировки глобальной переменной. Возвращается массив, содержащий выполняемый при трассировке код.

TracePoint (ruby 2.0)

Класс предназначен для замены `Kernel.set_trace_func` в объектном стиле. С помощью его экземпляров можно легко собирать информацию о процессе выполнения программы.

Создание трассировщика

::new(*events) { |a_trace_point| } *# -> a_trace_point*

Метод используется для создания объекта. Трассировка не начнется до тех пор пока не будет запущена в явной форме.

По умолчанию будут отслеживаться все доступные события. Для фильтрации событий могут использоваться следующие идентификаторы:

- *line:* выполнение новой строки кода;
  class: начало определения модуля или класса;
  end: конец определения модуля или класса;
  call: вызов метода;
  return: возвращение результата выполнения метода;
  c_call: вызов Си подпрограммы;
  c_return: возвращение результата выполнения Си подпрограммы;
  raise: получение исключения;
  b_call: начало выполнения блока;
  b_return: конец выполнения блока;
  thread_begin: начало выполнения потока;
  thread_end: конец выполнения блока.

```
trace = TracePoint.new(:call) do |tp|
    p [tp.lineno, tp.defined_class, tp.method_id, tp.event]
end
# -> #<TracePoint:0x007f17372cdb20>
```

- Отсутствие блока считается исключением ThreadError.
- Вызов методов, бесполезных для отслеживаемых событий, считается исключением RuntimeError.

```
TracePoint.trace(:line) do |tp|
    p tp.raised_exception
end
# -> RuntimeError!
```

- Вызов методов вне блока считается исключением RuntimeError.

```
TracePoint.trace(:line) do |tp|
  $tp = tp
end
$tp.line # -> RuntimeError!
```

::trace(*events) { |a_trace_point| } *# -> a_trace_point*

Версия предыдущего метода, автоматически запускающая трассировку.

Управление трассировкой

.enable *# -> boolean*

```
{ } # -> self
```

Метод используется для запуска трассировки. Когда трассировка не выполняется возвращается false.

```
trace.enabled?  # -> false
trace.enable    # -> false
trace.enabled?  # -> true
trace.enable    # -> true
```

Когда методу передается блок, то трассировка выполняется только в теле блока.

```
trace.enabled? # -> false

trace.enable do
  trace.enabled?
end

trace.enabled? # -> false
```

.disable *# -> boolean*

```
{ } # -> self
```

Метод используется для прекращения трассировки. Когда трассировка не выполняется возвращается false.

```
trace.enabled? # -> true
trace.disable  # -> false
trace.enabled? # -> false
trace.disable  # -> false
```

Когда методу передается блок, то трассировка прекращается только во время выполнения блока.

```
trace.enabled? # -> true

trace.disable do
  trace.enabled?
end

trace.enabled? # -> true
```

.enabled? *# -> boolean*

Метод используется для проверки статуса активности трассировки.

Интроспекция

.binding *# -> a_binding*

Метод используется для сохранения текущего состояния выполнения программы.

.defined_class *# -> a_module*

Метод используется для получения ссылки на класс или модуль (возвращается собственный класс объекта), в котором был вызван обрабатываемый метод. Это позволяет выполнять интроспекцию состояния.

```
class C; def foo; end; end
trace = TracePoint.new(:call) do |tp|
  p tp.defined_class # -> C
end.enable do
  C.new.foo
end
```

```
module M; def foo; end; end
class C; include M; end;

trace = TracePoint.new(:call) do |tp|
  p tp.defined_class # -> M
end.enable do
  C.new.foo
end
```

```
class C; def self.foo; end; end
trace = TracePoint.new(:call) do |tp|
  p tp.defined_class # -> #<Class:C>
end.enable do
  C.foo
end
```

.event *# -> symbol*

Метод используется для получения типа события.

.inspect *# -> string*

Метод используется для получения текстового сообщения о состоянии объекта.

.lineno *# -> integer*

Метод используется для получения номера строки кода.

.method_id *# -> string*

Метод используется для получения имени вызванного метода.

.path *# -> string*

Метод используется для получения пути к выполняемому файлу.

.raised_exception *# -> an_exception*

Метод используется для получения экземпляра вызванного исключения (только для события `:raise`).

.return_value *# -> object*

Метод используется для получения возвращенного результата (только для событий `:return`, `:c_return`, `:b_return`).

.self *# -> a_trace_point*

Метод используется для получения трассировщика, отслеживающего событие.

Глава 20

Конкуренция и параллелизм

20.1. Основы

Конкуренция в Ruby может быть реализована с помощью потоков выполнения или сопрограмм. Параллелизм может быть реализован с помощью процессов.

20.1.1. Параллелизм

Процесс - максимальная единица планирования ядра ОС. Ресурсы для процессов выделяются системой (каждый процесс использует отдельные ресурсы). При запуске программы создается новый процесс, в пространстве которого она выполняется. В связи с этим процессом также иногда называют непосредственное выполнение кода программы.

Поток выполнения - это минимальная единица планирования. Внутри каждого процесса существует по крайней мере один поток выполнения. Потоки выполнения, находящиеся внутри одного процесса совместно используют его адресное пространство и состояние выполнения.

Каждый раз запуская приложение создается отдельный процесс. Внутри каждого процесса создается основной поток выполнения, в котором выполняется код программы.

Потоки выполнения (thread) легко перепутать с потоками данных (io). Поток выполнения - это упорядоченная группа выполняемых действий (инструкций, выражений, команд и т.д.), а поток данных - это упорядоченная группа данных, которые могут быть записаны или прочитаны.

Идея распараллеливания вычислений основана на том, что некоторые задачи (процессы) могут быть разделены на группы меньших задач (потоков выполнения), выполняющиеся одновременно.

Параллельное программирование включает черты последовательного, но имеет три дополнительных этапа.

- Разбиение программы на несколько подпрограмм, которые могут выполняться одновременно;
- Изменение структуры программы для эффективного выполнения подпрограмм;
- Реализация параллелизма в исходном коде.

Создание программ для параллельного выполнения сложнее, чем для последовательного из-за возникновения конкуренции за ресурсы. Взаимодействие и

синхронизация между процессами повышают сложность проектирования параллельных программ.

Параллельные вычисления стали доминирующей парадигмой после появления многоядерных процессоров.

Использование потоков облегчает распараллеливание, так как переключение между потоками легче, чем переключение между процессами, и потоки используют единое адресное пространство. Это обеспечивает быстрый доступ к глобальным данным, но одновременно возникает риск их непреднамеренного искажения (чтобы этого не происходило, необходимо соблюдать некоторую дисциплину программирования).

20.1.2. Конкуренция

В Ruby реализована поддержка потоков, но при этом применяется глобальная блокировка интерпретатора (GIL), запрещающая выполнение двух или более потоков одновременно. GIL фактически ограничивает параллелизм на многоядерных системах, заменяя его конкуренцией.

Конкуренция - это широко распространенная модель проектирования и выполнения кода, позволяющая нескольким потокам выполняться в рамках одного процесса. Процессор переключается между разными потоками выполнения. Это переключение выполняется достаточно часто, чтобы восприниматься как одновременное выполнение.

Глобальная блокировка необходима, потому что управление памятью в Ruby реализовано в расчете на последовательное выполнение программ и не подразумевает встроенной защиты от изменения различными потоками (а это не безопасно).

Реализация конкуренции в Ruby позволяет реагировать на ввод данных. Обычно если основной поток выполнения заблокирован, то выполнение программы останавливается. Вместо этого, после блокировки основного потока, позволяется переключаться на выполнение производного потока, избегая ожидания реакции системы.

Блокировка потока отличается от завершения выполнения. После блокировки выполнение потока может быть продолжено с места выполнения блокировки (точки останова). Таким образом блокировка потока - это ожидание, а не завершение.

20.1.3. Состояние гонки

Если разные потоки могут изменить одну и ту же переменную, то они потенциально могут достичь состояния гонки. Состояние гонки - одно из главных препятствий параллельного программирования.

Большинство выражений выполняется в несколько операций. Выполнив первую операцию один из потоков может приступить к выполнению следующей операции, в то время как другой поток, выполняя первую операцию, влияет на состояние первого потока.

```
register = i
register = register + 1
i = register
```

В то время как первый поток приступает к третьей операции, второй поток может изменить значение переменной, выполняя вторую операцию.

20.1.4. Современный параллелизм

Реализация параллелизма в Ruby считается устаревшей на фоне повального распространения многоядерных систем. В других языках программирования существует несколько более современных решений.

- Атомарные инструкции: превращение выражений в атомарные операции;
- Транзакционная память (STM): гарантирует что выражения, выполняемые в одной транзакции, будут атомарными (Clojure);
- Акторы: проектирование кода так, что только один поток может изменять переменную. Выполнение каждого потока в отдельной области видимости, взаимодействие с этими потоками с помощью передачи сообщений (Scala, Erlang).

20.2. Потоки выполнения (Thread)

> *Хотя кажется, что потоки выполнения - это небольшой шаг от последовательных вычислений, по сути они представляют собой огромный скачок. Они отказываются от наиболее важных и привлекательных свойств последовательных вычислений: понятности, предсказуемости и детерминизма. Потоки выполнения, как модель вычислений, являются потрясающе недетерминированными, и работа программиста становится одним из обрезков этого недетерминизма.*
>
> *Edward A. Lee*

Поток выполнения, создаваемый при запуске программы, называется основным, а потоки выполнения, создаваемые разработчиком - производными.

После завершения основного потока выполнения, процесс выполнения программы прекращается, даже если производные потоки еще не выполнены.

Производные потоки используют ту же глобальную область видимости, что и основной, но многие глобальные переменные отличаются для разных потоков.

Каждый поток операционной системы имеет свой собственный стек, который занимает несколько килобайт памяти, которые, будучи умноженными на количество одновременных соединений, могут занять несколько сот мегабайт. Но если с потерей памяти можно смириться (она стоит дешево), то вычислительные затраты на создание и закрытие потоков, на переключение контекста и на синхронизацию, будут весьма заметны.

Потоки выполнения в Ruby оправдано применять только если существует несколько больших потоков данных (или, в общем, долгое ожидание реакции системы).

Потоки могут быть равносильны объектам - они также инкапсулируют некоторое состояние и обмениваются сообщениями для его изменения.

Состояние потоков:

- *Выполняющийся* - поток, выполнение которого еще не завершено;
  Ожидающий - поток, бездействующий до выполнения определенного условия (ограничения по времени, получения ответа от системы);
  Выполненный - поток, выполнение которого завершено. Выполнение может быть завершено нормально, с ошибкой или преждевременно.

Потоки выполнения создаются выполняющимися. Текущий поток выполняется до тех пор пока не будет переведен в режим ожидания или выполнен. После этого начинается выполнение следующего потока. Переключение между потоками совершается при выполнении системных вызовов (например ввода/вывода).

20.2.1. Thread

Класс Thread реализует стандарт POSIX для реализации потоков выполнения.

::start(*arg) { |*arg| } *# -> thread*

Синонимы: `fork, new`

Используется для создания нового потока.

```
Thread.start { } # -> #<Thread:0x962817c run>
```

::list *# -> array*

Массив, содержащий все созданные потоки выполнения.

```
Thread.list # -> [#<Thread:0x94da0a4 run>, #<Thread:0x964b4d8 sleep>!
```

::current *# -> thread*

Текущий поток выполнения.

```
Thread.current # -> #<Thread:0x94da0a4 run>
```

::main *# -> thread*

Основной поток выполнения.

```
Thread.main # -> #<Thread:0x94da0a4 run>
```

Управление текущим потоком

::pass *# -> nil*

Используется для переключения потоков. Переключение выполняется в зависимости от операционной системы и процессора (т.е. не обязательно).

```
Thread.start { Thread.pass } # -> #<Thread:0x962e554 run>
```

::stop *# -> nil*

Используется для переключения текущего потока выполнения в режим ожидания.

::exit *# -> thread*

Синонимы: `kill`

Используется для завершения выполнения текущего потока. Если выполнение уже завершено, то возвращается ссылка на класс Thread.

Управление произвольным потоком

.join(sec = nil) *# -> thread*

Используется для переключения текущего потока в режим ожидания до тех пор пока не будет выполнен поток, для которого метод был вызван (выполнение потока может быть приостановлено через переданное количество секунд - в этом случае возвращается nil).

```
a = Thread.new { print ?a; sleep(10); print ?b; print ?c }
x = Thread.new { print ?x; Thread.pass; print ?y; print ?z }
x.join
# -> "axyz"

y = Thread.new { 4.times { sleep 0.1; puts 'tick... ' } }
puts "Waiting" until y.join 0.15
# ->
"tick...
Waiting
tick...
Waiting
tick...
tick..."
```

Во второй версии Ruby вызов метода для текущего или основного потоков считается исключением `ThreadError`.

.value *# -> object*

Используется для выполнения потока (с помощью метода `.join`).

.run *# -> self*

Используется для переключения потока в режим выполнения. Текущий поток при этом переводится в режим ожидания, после чего начинается выполнение процесса, для которого метод был вызван.

.wakeup *# -> self*

Используется для переключения потока в режим выполнения (при этом поток может быть заблокирован).

.exit(status) *# -> self*

Синонимы: `kill, terminate`

Используется для завершения выполнения потока с переданным статусом. Если выполнение уже завершено, то возвращается ссылка на класс Thread.

.priority *# -> integer*

Приоритет выполнения потока. Основной поток выполняется с нулевым приоритетом. Производные потоки, наследуют приоритет от базовых.

Потоки выполнения с большим приоритетом выполняются раньше, чем потоки с меньшим приоритетом. Работа данного метода зависит от операционной системы.

.priority=(integer) *# -> self*

Используется для изменения приоритета выполнения для потока.

.add_trace_func(proc = nil) *# -> proc*

Синонимы: `set_trace_func`

Используется для обработки изменения состояния потока с помощью переданной подпрограммы. Если передается nil, то перехват выполнения прекращается.

Локальные переменные

.[name]=(object) *# -> object*

Метод используется для инициализации локальных переменных. Переменные будут существовать только для сопрограмм, выполняемых в теле потока (тело потока считается базовой сопрограммой).

```
[
  Thread.new { Thread.current["name"] = "A" },
  Thread.new { Thread.current[:name]  = "B" },
  Thread.new { Thread.current["name"] = "C" }
].each do |th|
  th.join
  puts "#{th.inspect}: #{th[:name]}"
end

# ->
#<Thread:0x00000002a54220 dead>: A
#<Thread:0x00000002a541a8 dead>: B
#<Thread:0x00000002a54130 dead>: C
```

```
Thread.main[:local] = 4 # -> 4
```

.[name] *# -> object*

Значение локальной переменной. Если переменная не существует, то возвращается nil.

```
Thread.main[:local] # -> 4
```

.keys *# -> array*

Массив идентификаторов всех локальных переменных.

```
Thread.main.keys # -> [:local]
```

.key?(name)

Проверка существования локальной переменной.

```
Thread.main.key? :global # -> false
```

Ruby 2.0

Во второй версии Ruby добавлены методы для работы с локальными переменными потока. В отличии от метода `.[]`, локальные переменные потока будут сохранять свое значение в теле сопрограмм.

.thread_variable_set(name, value)

Метод используется для инициализации локальных переменных потока.

.thread_variable_get(name) *# -> value*

Метод используется для получения значения локальной переменной потока.

```
Thread.new {
  Thread.current.thread_variable_set("foo", "bar")
  Thread.current["foo"] = "bar"

  Fiber.new {
    Fiber.yield [
      Thread.current.thread_variable_get("foo"), # -> 'bar'
      Thread.current["foo"],                     # -> nil
    ]
  }.resume
}.join.value # -> ['bar', nil]
```

.thread_variable?(name) *# -> bool*

Метод используется для проверки существования локальной переменной потока.

```
me = Thread.current
me.thread_variable_set(:oliver, "a")
me.thread_variable?(:oliver)    # -> true
me.thread_variable?(:stanley)   # -> false
```

.thread_variables *# -> array*

Метод используется для получения массива имен локальных переменных потока.

```
thr = Thread.new do
  Thread.current.thread_variable_set(:cat, 'meow')
  Thread.current.thread_variable_set("dog", 'woof')
end
thr.join               # -> #<Thread:0x401b3f10 dead>
thr.thread_variables   # -> [:dog, :cat]
```

Обработка ошибок

::abort_on_exception *# -> bool*

Глобальные настройки по обработке исключений. По умолчанию - false.

::abort_on_exception=(bool) *# -> bool*

Используется для применения режима отладки. При получении исключения в любом из производных потоков, выполнение основного потока прекращается. Подобное поведение также применяется если `$DEBUG` ссылается на true (программа запущена с ключом `-d`).

Основной поток завершается с помощью выражения `Thread.main.exit(0)`.

.abort_on_exception *# -> bool*

Глобальные настройки по обработке ошибок. По умолчанию - false.

.abort_on_exception= (bool) *# -> bool*

Используется для применения режима отладки. При получении исключения в любом из производных потоков, выполнение основного потока прекращается. Подобное поведение также применяется если `$DEBUG` ссылается на true (программа запущена с ключом `-d`).

.raise(message = nil)

```
( exc, message = nil, pos = nil )
```

Возбуждение исключения для потока. Метод не может быть вызван для текущего потока выполнения.

Асинхронная обработка событий [ruby 2.0]

Во второй версии Ruby добавлены методы для асинхронной обработки прерываний. В качестве прерываний рассматриваются исключения (`.raise`), закрытие потока (`.kill`) и закрытие основного потока (все производные потоки также будут закрыты).

::handle_interrupt(options) { } *# -> result*

Метод используется для изменения поведения при обработке прерываний (для текущего потока). Переданный массив содержит классы исключений, ассоциируемые с идентификаторами выполняемых действий. Каждое исключение будет обработано с помощью переданного блока.

Действия:

:immediate - немедленный вызов прерывания;
:on_blocking - вызов прерывания только после блокировки потока;
:never - отмена вызова прерывания.

```
# Для основного потока вызов `RuntimeError` будет игнорироваться.
th = Thread.new do
  Thread.handle_interrupt(RuntimeError => :never) {
    begin
      # You can write resource allocation code safely.
      Thread.handle_interrupt(RuntimeError => :immediate) {
        # ...
      }
    ensure
      # You can write resource deallocation code safely.
    end
  }
end
Thread.pass
# ...
th.raise "stop"
```

::pending_interrupt?(exception_klass = nil) *# -> boolean*

Метод используется для проверки существования обработчиков прерываний в очереди (поток ожидает начало обработки прерывания). Когда методу передается аргумент, то проверяются только обработчики определенного класса прерываний.

```
th = Thread.new{
  Thread.handle_interrupt(RuntimeError => :on_blocking){
    loop do
      # ...
      # Безопасная точка для вызова прерывания (поток заблокирован).
      if Thread.pending_interrupt?
        Thread.handle_interrupt(Object => :immediate){}
      end
      # ...
    end
  }
}
# ...
th.raise # остановка потока.
```

.pending_interrupt?(exception_klass = nil) *# -> boolean*

Версия предыдущего метода для экземпляров класса.

Остальное

.group *# -> thgroup*

Группа, в которую входит поток выполнения. Если поток не входит ни в одну из существующих групп, то возвращается nil.

```
Thread.main.group # -> #<ThreadGroup:0x94d9d98>
```

.inspect *# -> string*

Информация об объекте.

```
Thread.main.inspect -> "#<Thread:0x94da0a4 run>"
```

.status *# -> object*

Статус выполнения потока.

- *"run"* - выполняющийся;
  "sleep" - ожидающий;
  false - выполненный;
  nil - выполненный с ошибкой;
  "aborting" - выполненный преждевремменно;

```
Thread.main.status -> "run"
```

.alive? *# -> bool*

Проверка будет ли выполняться поток (поток не выполнен).

```
Thread.main.alive? -> true
```

.stop? *# -> bool*

Проверка остановлено ли выполнение потока (поток не выполняется).

```
Thread.main.stop? -> false
```

.safe_level *# -> integer*

Уровень безопасности.

```
Thread.main.safe_level -> 0
```

.backtrace *# -> array*

Состояние выполнения потока.

.backtrace_locations(*args) *# -> array || nil [Ruby 2.0]*

Метод используется для получения стека выполнения текущего потока. Эквивалентно `.caller_locations`.

20.2.2. Группировка потоков (ThreadGroup)

Несколько потоков могут быть объединены в одном составном объекте - группе потоков. Каждый поток выполнения может одновременно входить только в одну группу. При добавлении потока в другую группу, он автоматически удаляется из текущей группы. Производные потоки выполнения входят в ту же группу, что и базовые.

::Default *# -> thgroup*

Группа, создаваемая по умолчанию. Основной поток выполнения будет относиться к этой группе.

::new *# -> thgroup*

Используется для создания новой группы.

```
ThreadGroup.new # -> #<ThreadGroup:0x9711728>
```

.add(thread) *# -> thgroup*

Используется для добавления потока выполнения в группу.

```
Thread.main.group # -> #<ThreadGroup:0x94d9d98>
ThreadGroup.new.add Thread.main # -> #<ThreadGroup:0x96f0154>
Thread.main.group # -> #<ThreadGroup:0x96f0154>
```

.enclose *# -> thgroup*

Используется для блокирования группы, запрещая добавлять или удалять содержащиеся в ней потоки выполнения.

```
Thread.main.group.enclose # -> #<ThreadGroup:0x96f0154>
ThreadGroup.new.add Thread.main # -> error
```

.enclosed? *# -> bool*

Проверка заблокирована ли группа.

```
Thread.main.group.enclosed? # -> true
```

.list *# -> array*

Массив потоков выполнения, входящих в группу.

```
Thread.main # -> #<Thread:0x94da0a4 run>
Thread.main.group.list # -> [#<Thread:0x94da0a4 run>]
```

20.2.3. Синхронизация потоков (Mutex)

Синхронизация потоков требуется, чтобы избежать ошибок при использовании потоками одних и тех же данных или устройств (состояние гонки при обновлении данных).

Один из главных принципов безопасного выполнения потоков: изменение общих данных должно выполняться атомарно. К сожалению, в Ruby, для выполнения атомарных операций существует только один механизм синхронизации - блокировка (Mutex - mutual exclusion, взаимное исключение). Блокировка гарантирует, что только один поток способен выполнять этот код в единицу времени.

Многопоточное программирование имеет такую плохую репутацию потому что блокировки очень сложно использовать на практике. Блокировки экспоненциально повышают сложность многопоточного программирования, поэтому Matz рекомендует использовать процессы вместо потоков.

Поток выполнения, который в настоящее время работает с данными, блокирует экземпляр класса Mutex. После выполнения работы, блокировка снимается. В зависимости от наличия блокировки изменяется реакция конкурирующих потоков.

::new *# -> mutex*

Используется для создания нового объекта.

.lock *# -> mutex*

Используется для блокировки. Если объект уже заблокирован текущим потоком, то посылается исключение `ThreadError`.

Во второй версии Ruby вызов метода при обработке сигнала (`.trap`) считается исключением `ThreadError`.

.try_lock *# -> bool*

Используется для блокировки объекта. Возвращается результат блокировки.

Во второй версии Ruby вызов метода при обработке сигнала (`.trap`) считается исключением `ThreadError`.

.synchronize *# -> bool*

Используется для блокировки объекта во время выполнения блока.

```
@mutex = Mutex.new
@mutex.synchronize do
  i += 1
end
```

Во второй версии Ruby вызов метода при обработке сигнала (`.trap`) считается исключением `ThreadError`.

.locked? *# -> bool*

Проверка заблокирован ли объект.

.owned? *# -> bool [Ruby 2.0]*

Был ли объект заблокирован текущим потоком?

.unlock *# -> mutex*

Используется для снятия блокировки. Если объект заблокирован другим потоком выполнения, то посылается исключение `ThreadError`.

Во второй версии Ruby вызов метода при обработке сигнала (`.trap`) считается исключением `ThreadError`.

.sleep(sec = nil) *# -> sec*

Используется для снятия блокировки и переключения в режим ожидания. Если объект заблокирован другим потоком выполнения, то посылается исключение `ThreadError`.

Во второй версии Ruby вызов метода при обработке сигнала (`.trap`) считается исключением `ThreadError`.

Также стоит помнить что ожидание может быть завершено до окончания таймера (например в результате получения сигнала).

20.3. Процессы (Process)

20.3.1. Linux

Процессы бывают системными и пользовательскими, так как все, что происходит в системе может быть запущено либо пользователем, либо ядром.

Процесс init вызывается по окончании загрузки системы и является базовым для всех пользовательских процессов.

Интерактивные процессы связаны с терминалом, посредством которого можно взаимодействовать с процессом (посредством сигналов или ввода/вывода данных).

Демоны (неинтерактивные процессы) не связаны с терминалом и могут взаимодействовать с системой только с помощью сигналов.

Зомби - это процессы, зависшие в режиме ожидания.

Свойства процесса:

- *PID* - идентификатор процесса;
- *PPID* - идентификатор базового процесса;
- *UID* - реальный идентификатор владельца (обычно это пользователь, запустивший процесс);
- *EUID* - действующий идентификатор владельца. Определяет права доступа к файлам для процесса;
- *GUID* - реальный идентификатор группы владельцев;
- *EGUID* - действующий идентификатор группы владельцев. Наследует права доступа для группы, к которой принадлежит запустивший процесс пользователь;

- имя владельца процесса;
- приоритет;
- терминал.

Стадии жизненного цикла процесса:

- запуск (рождение);
- выполнение;
- завершение (смерть);

Вытеснение одной программы другой в рамках единственного процесса - типичный способ запуска новой программы. На самом деле производные процессы - это замещение копий базовых процессов.

20.3.2. Системные команды

`ps` - системная команда, выводящая информацию о процессах.
Сигналы:

- Корректное завершение: kill -15 PID или kill -TERM PID
- Принудительное завершение: kill -9 или kill -KILL

Изменение приоритета:

- Повышение: nice -5 NAME (по умолчанию 10);
- Понижение: nice –7 NAME;
- Изменение: renice 7 PID

Пользователь может управлять только теми процессами, хозяином которых он является, а администратор способен управлять любыми процессами, в том числе и демонами.

20.3.3. Ruby

Процесс - это программа в стадии ее выполнения. Ruby позволяет манипулировать процессами, используя низкоуровневые возможности системы.

Использование процессов позволяет достигать параллельного выполнения кода на многоядерных системах, но увеличивает сложность программы и количество потребляемой памяти.

.fork { nil } *# -> status*

Используется для выполнения системного вызова, создающего новый процесс, который является копией процесса, выполняющего этот вызов.

- PID процессов отличаются;
- PPID производного процесса равен PID базового;

- Для производного процесса создается собственная таблица файловых дескрипторов, копирующая таблицу базового процесса. Изменения не синхронизируются.

Подпроцессы обычно используются для выполнения системных вызовов exec, загружающих в пространство подпроцесса новую программу. Однако ничто не мешает использовать подпроцессы для выполнение параллельных задач.

В Linux страницы памяти базового процесса копируются производным, только после их изменения. Это позволяет уменьшить время создания процессов и количество потребляемой памяти.

Для производного процесса копируется только текущий поток выполнения.

Переданный блок выполняется в теле производного процесса. После выполнения производного процесса блок закрывается и возвращается 0.

В другом случае код программы после вызова метода, выполняется дважды - для базового процесса и для производного процесса:

- Для базового процесса в результате вызова метода возвращается идентификатор производного процесса;
- Для производного процесса в результате вызова метода возвращается nil.

Базовый процесс должен обрабатывать статусы завершения производных процессов с помощью методов `::wait` или `::detach`, иначе процессы могут превратиться в зомби.

Если создание подпроцессов не реализовано для ОС, то выполнение `Process.respond_to?(:fork)` вернет false.

.fork { nil } *# -> status [PRIVATE: Kernel]*

Версия предыдущего метода из модуля Kernel.

20.4. Обработка сигналов (Signal)

Сигналы - это способ передачи сообщений между процессами.

::list *# -> hash*

Массив названий сигналов, ассоциируемых с цифровыми кодами.

::trap(name, command) *# -> object*

```
(name) { } # -> object
```

Используется для обработки сигнала после его получения. Метод принимает либо название сигнала (приставка SIG может быть пропущена), либо его цифровой код. В результате возвращается предыдущий обработчик.

Во второй версии Ruby обработка `:SEGV`, `:BUS`, `:ILL`, `:FPE`, `:VTALRM` считается исключением `ArgumentError`.

command:

- блок, выполняемый при получении сигнала;
- текст:
  - `"IGNORE" ("SIG_IGN")` - полученный сигнал игнорируется;
    `"DEFAULT" ("SIG_DFL")` - сигнал обрабатывается как обычно;
    `"SYSTEM_DEFAULT"` - сигнал обрабатывается в зависимости от операционной системы;
    `"EXIT" (0)` - завершение выполнения программы.
- системный вызов:
  - `string` - текст команды для используемой оболочки: по умолчанию в Unix - это `"/bin/sh"`, а в Windows - `ENV["RUBYSHELL"]` или `ENV["COMSPEC"]`;
    `string, *arg` - текст команды и передаваемые аргументы;
    `{ [string, first_arg], *arg }` - текст команды, первый аргумент и остальные аргументы.

.trap(name, command) *# -> object [PRIVATE: Kernel]*

`(name) { } # -> object`

Версия предыдущего метода из модуля Kernel.

::signame(signo) *# -> string [Ruby 2.0]*

Метод используется для получения названия сигнала с переданным номером.

```
Signal.trap("INT") { |signo| puts Signal.signame(signo) }
Process.kill "INT", 0

# -> 'INT'
```

20.5. Событийная модель

Создание процессов очень дорого, поэтому для асинхронного IO лучше использовать потоки выполнения или даже события.

Событийная модель обходит ограничения неблокирующего IO. Такая модель очень хорошо масштабируется вертикально, но не горизонтально. После начала IO поток выполнения приостанавливается. После завершения IO возникает событие.

Событийная модель реализована в Goliath и Thin.

Для реализации событийной модели хорошо подходят сопрограммы. К сожалению при чтении/записи с диска событий о завершении не вызывается (вместо этого обычно используется мультиплексирование).

20.6. Сбор мусора

Для того, чтобы лучше понять особенности реализации сбора мусора, необходимо глубже погрузиться в Си реализацию интерпретатора.

Все объекты хранятся в виде структур RArray, RHash, RFile и т.д. Каждая структура представляет собой набор данных и набор флагов, обрабатываемых интерпретатором. Общее название для всех структур - RValue.

Интерпретатор размещает и организует RValue в массиве, который также называется ”куча”. Для ускорения создания объектов структуры создаются заранее.

При выполнении произвольной программы, создание нового объекта сводится к поиску доступной структуры.

Проблемы:

- отсутствие свободных структур;
- отсутствие необходимости создавать новый объект (можно использовать уже существующий).

Когда свободные структуры заканчиваются запускается сбор мусора. Сборщик мусора освобождает структуры, которые больше не используются в программе (в программе нет ссылок на структуру).

Высокоуровневая работа сборщика мусора:

- пометка существующих структур - перебор всех переменных и ссылок в программе и пометка соответствующих структур флагом FL_MARK. Помеченные структуры не могут быть удалены или использованы заново;
- хранение списка не помеченных структур. Все новые объекты сохраняются в структурах из этого списка. При сохранении структура из списка удаляется. Как только структуры заканчиваются (список пуст) начинает работу сборщик мусора. Если все структуры заняты и используются, то создается новая ”куча” (фактически за один раз создается 10 ”куч”).

Основной недостаток такой реализации сборщика в том, что пометка структур вызывает копирование памяти для производных процессов.

Обычно данные базового процесса могут свободно использоваться производными процессами (это позволяет не копировать память и ускоряет создание процессов). При изменении общих данных создаются новые фрагменты памяти. Это позволяет иметь общую ”кучу” для нескольких процессов. Пометка структур, к сожалению, считается изменением общих данных, поэтому копирование памяти происходит постоянно.

20.6.1. Ruby 2.0

Новый алгоритм для сбора мусора называется ”Bitmap marking” (битовая маркировка). Алгоритм обещает снизить потребление памяти производными процессами (в особенности на веб-серверах, запускающих несколько копий одного приложения).

Впервые этот алгоритм был реализован в ree (ruby enterprise edition).

Вместо установки флага, создается отдельный массив битов, ссылающийся на структуры. Для каждой кучи создается массив, содержащий набор битов. 1 аналогична флагу FL_MARK.

Алгоритм реализован с помощью заголовков (первая структура в куче), ссылающихся на массив битов.

Это позволяет отмечать все используемые структуры без их действительного изменения (не устанавливая флаги), чтобы ядро могло грамотно распределять память между процессами. При этом маркировка выполняется нерекурсивно, позволяя избежать неожиданного переполнения стека.

Массивы битов конечно изменяются, но они содержат непрерывный поток битов и достаточно малы.

Важная особенность в том, что память, выделяемая для кучи, теперь должна быть согласована. Вместо malloc() вызывается posix_memalign().

Глава 21

Безопасность

Реализация собственной системы безопасности - одна из спорных и неоднозначных особенностей в Ruby.

Безопасность кода в Ruby обеспечивается с помощью применения различных модификаторов, запрещающих изменение значения объекта или маркирующих небезопасные данные.

Интерпретатор автоматически считает небезопасными:

- аргументы, переданные при запуске программы (элементы массива ARGV);
- переменные окружения (элементы массива ENV);
- любые данные, извлекаемые из файлов, сокетов или потоков;
- объекты, создаваемые на основе небезопасных данных.

`$SAFE` ссылается на значение текущего уровня безопасности. Переменная локальна для каждого отдельного блока кода или процесса. Уровень безопасности основного процесса объявляется при запуске программы с помощью ключа `-T` (по умолчанию - 0).

Нарушение ограничений безопасности считается исключением `SecurityError`.

21.1. Уровни безопасности

Более высокие уровни безопасности наследуют ограничения более низких.

21.1.1. Уровень 0

Не предполагает ограничений.

21.1.2. Уровень 1

Запрещается:

- загружать библиотеки, если их название небезопасно;
- выполнять произвольный код, если текст кода небезопасен;
- открывать файлы, если их название небезопасно;
- соединяться с хостом, если его название небезопасно;
- запускать программу с ключами `-e, -i, -l, -r, -s, -S, -X`;

- выполнять методы из классов Dir, IO, File, FileTest, передавая им небезопасные аргументы;
- выполнять методы `test`, `eval`, `require`, `load`, `trap`, передавая им небезопасные аргументы.

Также игнорируются переменные окружения RUBYLIB и RUBYOPT и текущий каталог при поиске подключаемых библиотек.

21.1.3. Уровень 2

Запрещается выполнять методы

- `fork`
- `syscall`
- `exit!`
- `Process.equid=`
- `Process.fork`
- `Process.setpgid`
- `Process.setsid`
- `Process.kill`
- `Process.seeprioprity`

передавая им небезопасные аргументы.

21.1.4. Уровень 3

Все объекты, кроме предопределенных считаются небезопасными. Вызов метода `.untaint` запрещается.

21.1.5. Уровень 4

Запрещается:

- изменять значение безопасных объектов;
- загружать библиотеки с помощью `require` или `load` (разрешается в теле анонимного модуля);
- использовать метапрограммирование;
- работать с любыми процессами кроме текущего;
- завершать выполнение процесса;
- читать или записывать данные;
- изменять переменные окружения;
- вызывать методы `.srand` и ‘Random.srand.

Разрешается вызывать метод `.eval`, передавая ему небезопасные аргументы.

21.2. Модификаторы

Модификаторы в данном случае объявляются с помощью методов.

.freeze *# -> self*

Запрещает изменять значение объекта.

.frozen? *# -> bool*

Проверка запрещено ли изменять значение объекта.

.taint *# -> self*

Используется для маркировки небезопасных данных. Этим модификатором отмечаются полученные внешние данные.

.untaint *# -> self*

Используется для маркировки безопасных данных.

.tainted? *# -> bool*

Проверка безопасно ли использование объекта.

.untrust *# -> self*

Используется для маркировки данных, которым разработчик не доверяет. Этим модификатором отмечаются данные, полученные при выполнении кода с уровнем безопасности 4.

.trust *# -> self*

Используется для маркировки данных, которым разработчик доверяет.

.untrusted? *# -> bool*

Проверка доверяет ли разработчик объекту.

Приложения

Приложение А

Запуск программы

Ключи:

--copyright - отображение сведений о копирайте;
--version - отображение версии интерпретатора;
-(-h)elp - отображение справочной информации;
-0 [codepoint] - изменение символа перевода строки, использующегося при чтении из потока (`$/`). Кодовая позиция задается в восьмеричной системе счисления.

- Если кодовая позиция не указана, то строки разделятся не будут;
- Если указана позиция -00, то в качестве разделителя будут использоваться два символа перевода строки подряд;
- Если указана позиция -0777, то файлы буду обрабатываться как одна большая строка.

-C (-X) <dir> - изменение базового каталога;
--encoding (-E) <external>[:internal] - изменение внешней и внутренней кодировок;
-F <pattern> - изменение разделителя частей текста (`$;`), использующегося при вызове метода `.split`. В качестве образца передаются произвольные символы или тело регулярного выражения;
-I <dirs> - добавление дополнительных каталогов. Данные каталоги будут добавлены в начало `$LOAD_PATH ($:)` и использованы для поиска необходимых библиотек. Каталоги разделяются двоеточием (Linux) или точкой с запятой (Windows);
-K <encoding> - изменение внешней и внутренней кодировок.

- *e* - EUC-JP;
  s - Windows-31J (CP932);
  u - UTF-8;
  n - ASCII-8BIT (BINARY).

-S - поиск программы с помощью переменной окружения PATH;
-T [security] - изменение уровня безопасности для программы (по умолчанию 1);
-U - использование UTF-8 в качестве внутренней кодировки;
-W [verbose] - изменение степени подробности отладочной информации:

- *0* - без предупреждений, `$VERBOSE` ссылается на nil;
  1 - средний уровень, `$VERBOSE` ссылается на false;
  2 (по умолчанию) - выводятся все предупреждения, `$VERBOSE` ссылается на true.

-a - автоматическое разделение кода на строки при использовании ключей `-n` или `-p`. В начале каждой итерации цикла выполняется код: `$F = $_.split!`;

-c - проверка синтаксиса. Если ошибок не найдено, то в стандартный поток для вывода передается `"Syntax OK"`;

-(-d)ebug - запуск в режиме отладки (`$DEBUG` ссылается на true).

Наличие ключа позволяет писать код для отладки программы, который будет выполняться только если `$DEBUG` ссылается на true;

`{verbatim} if $DEBUG`

-e <verbatim> - выполнение произвольного кода;

-i [ext] - запуск в режиме редактирования, позволяющий записывать данные с помощью ARGF. Расширение используется для создания резервных копий файлов;

-l - автоматическое изменение кода. При этом, во-первых, `$\` копирует `$/`, и, во-вторых, для каждой строки вызывается метод `.chop!`;

-n - выполнение программы в теле цикла (каждая строка программы выполняется отдельно):

```
while gets
  # code
end
```

-p - выполнение программы в теле цикла (каждая строка кода выполняется отдельно, результат передается в стандартный поток для вывода):

```
while gets
  # code
end
print
```

-r <lib> - загрузка дополнительной библиотеки перед выполнением с помощью `.require`;

-s - предварительная обработка аргументов, начинающихся с дефиса. Обработка выполняется до любого обычного аргумента или символов `--`. Обработанные аргументы будут удалены из ARGV.

- Для аргументов вида `-x=y`, `$x` в теле программы будет ссылаться на y;
- Для аргументов вида `-x`, `$x` в теле программы будет ссылаться на true.

-(-v)erbose - запуск программы с ключом -w. Если имя программы не указано, то отображается версия интерпретатора.

Наличие ключа позволяет писать код для отладки программы, который будет выполняться только если $VERBOSE ссылается на true;
{verbatim} if $VERBOSE
-w - отображение всех возможных предупреждений ($VERBOSE ссылается на true);
-x [dir] - отображение кода программы со строки, начинающейся с !# и содержащей ruby. Конец программы должен быть указан с помощью EOF (обозначение конца файла), ^D (control-D), ^Z (control-Z), или __END__. Переданный каталог используется вместо базового;
-(-y)ydebug - используется для отладки интерпретатора;
--disable-... (--enable-...) - включение или отключение указанной опции:

- *gems* - поиск библиотек в пакетах (отключение этой опции может ускорить запуск программы);
  rubyopt - использование переменной окружения RUBYOPT;
  all - выполнение двух перечисленных действий.

--dump <target> - используется для отладки интерпретатора.

Переменные окружения:

RUBYLIB - список каталогов, используемых для поиска библиотек;
RUBYOPT - список ключей, используемых для запуска программ. Могут быть добавлены только ключи -d, -E, -I, -K, -r, -T, -U, -v, -w, -W, --debug, --disable-... и --enable-...;
RUBYPATH - список каталогов, в которых выполняется поиск программы;
RUBYSHELL - путь к оболочке ОС. Переменная действительна только для mswin32, mingw32, и OS/2;
PATH - путь к интерпретатору.

Доступ к перемнным окружения может быть получен с помощью ENV - объекта, подобный ассоциативному массиву. Для объекта определен метод .to_hash, преобразующий его в настоящий ассоциативный массив.

Глобальные переменные:

$F - результат последнего выполнения выражения $_.split. Переменная определена, если программа запущена с ключами -a, -n или -p;
$-W - степень подробности предупреждений;
$-i - расширение, используемое для резервных копий;
$-d # -> bool ;
$-l # -> bool ;

$-v # -> bool ;
$-p # -> bool ;
$-a # -> bool ;
$* - массив аргументов, переданных при запуске программы;
$> - поток для записи по умолчанию;
$; ($-F) - используется при вызове метода `string.split`. Разделитель частей текста (по умолчанию nil);
$/ ($-0) - используется `kernel.gets`. Символ перевода строки (по умолчанию - `"\n"`). Если ссылается на nil, то вызов метода вернет весь текст;
$\ - разделитель, добавляемый при передаче объектов в поток (по умолчанию nil);
$, - используется при вызове методов `array.join` и `kernel.printf` Разделитель элементов (по умолчанию nil);
$~ - экземпляр класса MatchData для последнего поиска совпадений;
$& - текст последнего найденного совпадения;
$‘ - текст, предшествующий последнему найденным совпадением;
$’ - текст, следующий за последним найденным совпадением;
$+ - текст последнего совпадения с группой;
$1, $2, $3, $4, $5, $6, $7, $8, $9 - текст последнего совпадения с группой, имеющей соответствующий индекс;
$LOAD_FEAUTURES ($”) - массив всех использованных библиотек (изменение значения элементов запрещено в Ruby 2.0; для объектов не относящихся к String вызывается метод `object.to_path`);
$LOAD_PATH ($:, $-I) - массив каталогов, использующихся для поиска библиотек (изменение значения элементов запрещено в Ruby 2.0; для объектов не относящихся к String вызывается метод `object.to_path`);
$. - позиция последней извлеченной строки из потока;
$_ - последняя извлеченная строка из потока;
$! - последнее полученное исключение;
$@ - местоположение в коде последнего полученного исключения;
$DEBUG - true, если программа запущена с ключом `-(-d)ebug`;
$VERBOSE ($-v, $-w)

- *nil*, если программа запущена с ключом `-W0`;
  true, если программа запущена с ключами `-w` или `-(-v)erbose`;
  false в остальных случаях.

$$ - идентификатор текущего процесса;
$? - статус последнего выполненного процесса;
$FILENAME - относительный путь к текущему файлу в ARGF. При взаимодействии с стандартным потоком для ввода возвращается `"-"`;
$SAFE - текущий уровень безопасности.

Приложение B

Синтаксис регулярных выражений

Во второй версии Ruby для обработки регулярных выражений используется библиотека Onigmo: http://github.com/k-takata/Onigmo.

Также более подробное описание синтаксиса существует по адресу http://perldoc.perl.org/perlre.html.

Модификаторы:

i – поиск будет выполняться без учета регистра символов;

m – поиск будет выполняться в многострочном режиме. Точка в теле регулярного выражения будет соответствовать также и символу перевода строки;

x – пробельные символы (пробел, отступ, перевод строки) в теле регулярного выражения будут игнорироваться интерпретатором;

o – интерполяция в теле регулярного выражения будет выполняться только один раз, перед началом поиска;

u, e, s, n – тело регулярного выражения будет обрабатываться в указанной кодировке. Соответственно: u - UTF-8, e - EUC-JP , s - Windows-31J , n - ASCII-8BIT.

ASCII символы:

. - соответствует любому символу в тексте (кроме символа перевода строки в однострочном режиме поиска);

\w - соответствует любой букве, цифре или знаку подчеркивания;

\W - соответствует любому символу, кроме букв, цифр или знаков подчеркивания;

\s - соответствует любому пробельному символу (пробел, отступ, перевод строки);

\S - соответствует любому символу, кроме пробельных;

\d - соответствует любой десятичной цифре;

\D - соответствует любому символу, кроме десятичных цифр;

\h - соответствует любой шестнадцатеричной цифре;

\H - соответствует любому символу, кроме шестнадцатеричных цифр.

Unicode символы:

\R - **во второй версии Ruby** соответствует любому переводу строки, включая вертикальный отступ. Спецсимвол является аббревиатурой к non really.

Эквивалентно `(?>\x0D\x0A|[\x0A-\x0D\x{85}\x{2028}\x{2029}])` в Unicode или `(?>\x0D\x0A|[\x0A-\x0D])` в другом случае.

```
"\n" =~ /^\R$/   # -> 0
"\r" =~ /^\R$/   # -> 0
"\r\n" =~ /^\R$/ # -> 0
```

\X - **во второй версии Ruby** соответствует любому расширенному графическому символу в Unicode. Спецсимвол является аббревиатурой к eXtended Unicode character.

```
"P\u{307}" =~ /^\X$/ # -> 0
```

[[:класс:]] - соответствует любому символу, входящему в класс:

- *alnum* - буквы и цифры;
  alpha - буквы;
  ascii - ASCII символы;
  blank - пробел и отступ;
  cntrl - эмблемы составного текста;
  digit - десятичные цифры;
  graph - буквы, цифры и знаки препинания;
  lower - строчные буквы;
  print - буквы, цифры, знаки препинания и пробел;
  punct - знаки препинания;
  space - пробельные символы (пробел, отступ, перевод строки);
  upper - прописные буквы;
  word - буквы, цифры и специальные знаки препинания (знак подчеркивания);
  xdigit -шестнадцатеричные цифры;

\p{класс} - соответствует любому символу, входящему в класс.

\p{^класс} - соответствует любому символу, кроме входящих в класс.

- *Alnum* - буквы и цифры;
  Alpha - буквы;
  Any - Unicode символы;
  ASCII - ASCII символы;
  Assigned - свободные цифровые коды;
  Blank - пробел и отступ;
  Cntrl - эмблемы составного текста;
  Digit - десятичные цифры;
  Graph - буквы, цифры и знаки препинания;
  Lower - строчные буквы;
  Print - буквы, цифры, знаки препинания и пробел;
  Punct - знаки препинания;
  Space - пробельные символы (пробел, отступ, перевод строки);
  Upper - прописные буквы;
  Word - буквы, цифры и специальные знаки препинания (знак подчеркивания);
  Xdigit - шестнадцатеричные цифры.

Unicode-классы символов:

- C - остальные символы; Cc - спецсимволы; Cf - спецсимволы, влияющие на форматирование; Cn - свободные цифровые коды; Co - логотипы; Cs - символы-заменители;
- L - буквы; Ll - строчные буквы; Lm - особые символы; Lo - остальные символы; Lt - буквы в начале слова; Lu - прописные буквы;
- M – символы, использующиеся в связке; Mn - символы, изменяющие другие символы; Mc - специальные модификаторы, занимающие отдельную позицию в тексте; Me - символы, внутри которых могут находиться другие символы;
- N - цифры; Nd - десятичные цифры; Nl - римские цифры; No - остальные цифры;
- P - знаки препинания; Pc - специальные знаки препинания; Pd - дефисы и тире; Ps - открывающие скобки; Pe - закрывающие скобки; Pi - открывающие кавычки; Pf - закрывающие кавычки; Po - остальные знаки препинания;
- S - декоративные символы; Sm - математические символы; Sc - символы денежных единиц; Sk - составные декоративные символы; So - остальные декоративные символы;
- Z - разделители, не имеющие графического представления; Zs - пробелы; Zl - перевод строки; Zp - перевод параграфа.

Также можно указать класс, определяющий алфавит. Поддерживаемые алфавиты: Arabic, Armenian, Balinese, Bengali, Bopomofo, Braille, Buginese, Buhid, Canadian_Aboriginal, Carian, Cham, Cherokee, Common, Coptic, Cuneiform,

Cypriot, Cyrillic, Deseret, Devanagari, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Inherited, Kannada, Katakana, Kayah_Li, Kharoshthi, Khmer, Lao, Latin, Lepcha, Limbu, Linear_B, Lycian, Lydian, Malayalam, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Ol_Chiki, Old_Italic, Old_Persian, Oriya, Osmanya, Phags_Pa, Phoenician, Rejang, Runic, Saurashtra, Shavian, Sinhala, Sundanese, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Vai, and Yi.

Группировка символов:

[...] - соответствует любому символу из ограниченных квадратными скобками. Внутри квадратных скобок могут быть использованы диапазоны символов (a-z);

[^...] - соответствует любому символу, кроме ограниченных квадратными скобками;

(?:...) - символы объединяются в группу и используются как одна логическая единица;

(...) - символы объединяются в группу и используются как одна логическая единица. Группе будет присвоен порядковый номер (от 1 до 9);

(?<идентификатор>...) - символы объединяются в группу и используются как одна логическая единица. Группе будет присвоен указанный идентификатор.

Если лексема регулярного выражения использовалось в качестве левого операнда, тогда, после выполнения выражения, идентификаторы групп ссылаются на текст совпадения, или на nil, если совпадений не найдено. Идентификаторы групп объявляются как локальные переменные.

Глобальные переменные от $1 до $9 также ссылаются на текст совпадения с группой, имеющей указанный номер.

Группы символов:

\целое_число - соответствует тексту совпадения с группой, имеющей указанный номер;

\K <идентификатор> - соответствует тексту совпадения с группой, имеющей указанный идентификатор;

\g <...> - соответствует тексту совпадения с группой, имеющей указанный порядковый номер или идентификатор. **Во второй версии Ruby** добавлен расширенный синтаксис (см. документацию для Onigmo).

Повторы:

...? - соответствует от 0 до 1 повторам символа или группы;
...* - соответствует 0 и более повторам символа или группы. Результат поиска содержит максимально возможное совпадение (жадный алгоритм);
...*? - соответствует 0 и более повторам символа или группы. Результат поиска содержит минимально возможное совпадение (не жадный алгоритм);
...+ - соответствует 1 и более повторам символа или группы. Результат поиска содержит максимально возможное совпадение (жадный алгоритм);
...+? - соответствует 1 и более повторам символа или группы. Результат поиска содержит минимально возможное совпадение (не жадный алгоритм);
...{a, b} - соответствует от a до b повторам символа или группы. Результат поиска содержит максимально возможное совпадение (жадный алгоритм);
...{a, b}? - соответствует от a до b повторам символа или группы. Результат поиска содержит минимально возможное совпадение (не жадный алгоритм).
В обоих случаях допускается отсутствие a, b, или запятой.

Положение в тексте:

^... - соответствует символу или группе в начале строки;
...$ - соответствует символу или группе в конце строки;
\A... - соответствует символу или группе в начале текста;
...\z - соответствует символу или группе в конце текста;
...\Z - соответствует символу или группе в конце текста или перед последним символом перевода строки, замыкающим текст;
...\b - соответствует символу или группе в конце слова;
\b... - соответствует символу или группе в начале слова;
...\B - соответствует символу или группе в любом месте, кроме конца слова;
\B... - соответствует символу или группе в любом месте, кроме начала слова.

- Использование идиомы `/^...$/` для проверки полного совпадения текста с образцом потенциально может привести к небезопасному коду. Чтобы избежать этого, в данном случае, лучше использовать `/\A...\z/`.
- Идиома `\b...\b` часто используется для поиска отдельных слов в тексте.

Логические условия:

№1|№2 - объединение множеств;
№1&&№2 - пересечение множеств;
№1(?=№2) - соответствует символам №1, если символы №2 встречаются далее по тексту (позитивное заглядывание вперед);

№1(?!№2) - соответствует символам №1, если символы №2 не встречаются далее по тексту (негативное заглядывание вперед);

№1(?<=№2) - соответствует символам №1, если символы №2 встречаются в предыдущей части текста (позитивное заглядывание назад);

\K - **во второй версии Ruby** любой шаблон, находящийся слева от спецсимвола не будет добавлен к тексту совпадения (и соответственно не будет изменяться при замене совпадений).

Эквивалентно позитивному заглядыванию назад (`/№1 \K №2/` и `/(?<=№1) №2/`).

Спецсимвол является аббревиатурой к keep.

№1(?<!№2) - соответствует символам №1, если символы №2 не встречаются в предыдущей части текста (негативное заглядывание вперед).

(?(cond)yes|no) - **во второй версии Ruby** в зависимости от условия выполняется поиск совпадения с той или иной группой. Условием может быть совпадение с группой или идентификатор или даже заглядывание вперед/назад.

```
regexp = /^([A-Z])?[a-z]+(?(1)[A-Z]|[a-z])$/

regexp =~ "foo"   # -> 0
regexp =~ "foO"   # -> nil
regexp =~ "FoO"   # -> 0
```

Остальное:

(?#...) - игнорируемый комментарий;

(?>...) - в любом случае соответствует указанным символам;

(?№1-№2) - применяет модификаторы №1 и отменяет модификаторы №2 для дальнейшего поиска;

(?№1-№2:...) - применяет модификаторы №1 и отменяет модификаторы №2 для символов или групп.

Приложение C

Форматные строки

Форматная строка - это текст, в котором обычные символы перемешаны с специальными синтаксическими конструкциями. Обычные символы переносятся в результат без изменений, а специальные влияют на форматирование объектов.

Спецсимволы классифицируются по их влиянию на форматирование. Из них только устанавливающие тип форматирования являются обязательными.

Синтаксис спецсимволов (пробелы между ними не используются, а добавлены только для наглядности):

```
%[модификатор][размер][.точность]<тип форматирования>
```

Размер влияет на количество символов в результате. По умолчанию в начало текста добавляются дополнительные пробелы.

```
"%5d"%2 # -> "    2"
```

Точность влияет на количество цифр после десятичной точки (по умолчанию - 6).

Типы форматирования:

Для целых чисел:

b - преобразует число в десятичную систему счисления. Для отрицательных чисел будет использоваться необходимое дополнение до 2 с символами .. в качестве приставки.

```
"%b" % 2 # -> "10"
"%b" % -2 # -> "..10"
```

B - аналогично предыдущему, но с приставкой 0B в альтернативной нотации.

d (или i или u) - преобразует число из двоичной системы счисления в десятичную.

```
"%d" % 0x01 # -> "1"
```

o - преобразует число в восьмеричную систему счисления. Для отрицательных чисел будет использоваться необходимое дополнение до 2 с символами .. в качестве приставки.

```
"%o" % 2 # -> "2"
"%o" % -2 # -> "..76"
```

x - преобразует число в шестнадцатеричную систему счисления. Для отрицательных чисел будет использоваться необходимое дополнение до 2 с символами .. в качестве приставки.

```
"%x" % 2 # -> "2"
"%x" % -2 # -> "..fe"
```

X - аналогично предыдущему, но с приставкой 0X, в альтернативной нотации.

Для десятичных дробей:

e - преобразует число в экспоненциальную нотацию.

```
"%e" % 1.2 # -> "1.200000e+00"
```

E - аналогично предыдущему, но с использованием символа экспоненты E.

f - округление числа.

```
"%.3f" % 1.2 # -> "1.200"
```

g - преобразует число в экспоненциальную нотацию, если показатель степени будет меньше -4 или больше либо равен точности. В других случаях точность определяет количество значащих цифр.

```
"%.1g" % 1.2 # -> "1"
"%g" % 123.4 # -> "1e+02"
```

G - аналогично предыдущему, но с использованием символа экспоненты E.

a - преобразование числа в формат: знак числа, число в шестнадцатеричной системе счисления с приставкой 0x, символ p, знак показателя степени и показатель степени в десятичной системе счисления.

```
"%.3a" % 1.2 # -> "0x1.333p+0"
```

A - аналогично предыдущему, но с использованием приставки 0X и символа P.

Для других объектов:

c - результат будет содержать один символ.

```
"%c" % ?h # -> "h"
```

p - вызов метода `object.inspect` для объекта.

```
"%p" % ?h # -> "\"h\""
```

s - преобразование текста. Точность определяет количество символов.

```
"%.3s" % "Ruby" # -> "Rub"
```

Модификаторы:

- - пробелы будут добавляться не в начало, а в конец.

```
4b" % 2 # -> "10  "
```

0 (для чисел) - вместо пробелов добавляются нули.

```
"%04b" % 2 # -> "0010"
```

+ (для чисел) - результат будет содержать знак плюса для положительных чисел. Для oxXbB используется обычная запись отрицательных чисел.

```
"%+b" % -2 # -> "-10"
```

(для эмблем bBoxXaAeEfgG) - использование альтернативной нотации.

- Для o повышается точность результата до тех пор пока первая цифра не будет нулем, если не используется дополнительное форматирование.
  ```
  "%#o" % 2 # -> "02"
  ```
- Для xXbB используются соответствующие приставки.
  ```
  "%#X" % 2 # -> "0X2"
  ```
- Для aAeEfgG используется десятичная точка даже если в этом нет необходимости.
  ```
  "%#.0E" % 2 # -> "2.E+00"
  ```
- Для gG используются конечные нули.
  ```
  "%#G" % 1.2 # -> "1.20000"
  ```

* - соответствующий спецсимволу объект используется для определения размера. Для отрицательных чисел пробелы добавляются в конце результата.

```
"%*d" % [ -2, 1 ] # -> "1 "
```

Если форматная строка содержит несколько спецсимволов, то они будут последовательно использоваться для форматируемых объектов, которые должны храниться в индексном или ассоциативном массивах.

Для индексных массивов модификатор `цифра$` изменяет форматирование элемента с соответствующей позицией (начиная с 1). При этом модификатор должен быть указан для каждого спецсимвола.

```
"%3$d, %1$d, %1$d" % [ 1, 2, 3 ] # -> "3, 1, 1"
```

Для ассоциативных массивов модификатор `<идентификатор>` после приставки применяет форматирование для элемента с соответствующим ключом.

```
"%<two>d, %<one>d" % { one: 1, two: 2, three: 3 } # -> "2, 1"
```

Приложение D

Присваивание

Синтаксис выражения:

- Для одного идентификатора и одного объекта выполняется стандартное присваивание. Возвращается ссылка на объект.
  `x = ?R # -> "R"`
- Для нескольких идентификаторов и одного объекта, сначала вызывается метод `object.to_ary`, а затем в присваивании участвуют все полученные элементы. Возвращается ссылка на массив.

```
x, y  =  [ 1, 2 ] # -> [ 1, 2 ]
x # -> 1
y # -> 2
```

- Для одного идентификатора и нескольких объектов выполняется присваивание массива, содержащего все объекты. Возвращается ссылка на массив.

```
x  =  ?Y, ?N # -> [ "Y", "N" ]
x # -> "Y"
```

- Для одинакового количества идентификаторов и объектов выполняется параллельное присваивание. Возвращается ссылка на массив из всех использованных объектов.

```
x, y  =  ?Y, ?N # -> [ "Y", "N" ];
x # -> "Y"
y# -> "N"

x, y  =  y, x # -> [ "N", "Y" ]
x # -> "N"
y # -> "Y"
```

- Если за идентификатором следует запятая, то интерпретатор считает, что после нее указан еще один идентификатор. Этот идентификатор участвует в присваивании, даже если фактически не присутствует.

```
x,  =  ?Y, ?N # -> [ "Y", "N" ]
x# -> "Y"
```

- Когда идентификаторов больше, чем объектов, выполняется параллельное присваивание. Лишние идентификаторы объявляются, но не инициализируются. Возвращается ссылка на массив из всех использованных объектов.

```
x, y, z  =  ?Y, ?N  # -> [ "Y", "N" ]
x # -> "Y"
y # -> "N"
z # -> nil
```

- Когда идентификаторов меньше, чем объектов, выполняется параллельное присваивание только тех идентификаторов, для которых хватило объектов. Возвращается ссылка на массив из всех использованных объектов.

```
x, y  =  ?Y, ?N, ?Q # -> [ "Y", "N", "Q" ]
x # -> "Y"
y # -> "N"
```

- Если перед объектом стоит символ звездочки (*), то объект обрабатывается как составной. Интерпретатор использует для присваивания все элементы объекта (с помощью метода `object.to_splat`). Возвращается ссылка на массив из всех использованных объектов.

  Запись двух символов звездочки подряд считается исключением. При извлечении элементов вложенных массивов символ звездочки необходимо записывать перед каждым вложенным массивом.

```
x, y  =  *(1..3) # -> [ 1, 2, 3 ]
x # -> 1
y # -> 2
```

- Если перед идентификатором стоит символ звездочки (*), то интерпретатор использует для присваивания массив из всех лишних объектов. Возвращается ссылка на массив из всех использованных объектов.

```
*x, y  =  ?Y, ?N # -> [ "Y", "N" ]
x # -> [ "Y" ]
y # -> "N"

*x, y  =  ?Y, ?N, ?Q # -> [ "Y", "N", "Q" ]
x # -> [ "Y", "N" ]
y # ->"Q"

x, *y, z  =  ?Y, ?N, ?Q, ?R # -> [ "Y", "N", "Q", "R" ]
x # ->"Y"
y # ->[ "N", "Q" ]
z # ->"R"
```

- Для идентификаторов, ограниченных круглыми скобками одновременно выполняется несколько выражений присваивания:

  1. Группа обрабатывается как одна логическая единица;
  2. Интерпретатор извлекает идентификаторы из группы.

Возвращается ссылка на массив из всех использованных объектов.

```
x, (y, z) = ?Y, ?N  # -> [ "Y", "N" ]
  x = ?Y
  (y, z) = ?N
    y, z = ?N
      y = ?N
      z = nil
```

Приложение E

Преобразование кодировок

Принимаемые элементы:

replace: текст, использующийся для замены символов. По умолчанию используется `uFFFD` для символов Unicode и `?` для других символов;
:invalid => :replace - замена ошибочных байтов;
:undef => :replace - замена отсутствующих символов;
:fallback => encoding - изменение кодировки отсутствующих символов;
:xml => :text - экранирование символов из XML CharData. Результат может быть использован в HTML 4.0:

- & на `&`
- < на `<`
- > на `>`
- замена отсутствующих символов на байты вида `&x` (где x - цифра в шестнадцатеричной системе счисления).

:xml => :attr - экранирование символов из XML AttrValue. Результат выделяется двойными кавычками и может быть использован для значений свойств в HTML 4.0:

- & на `&`
- < на `<`
- > на `>`
- " на `"`
- замена отсутствующих символов на байты вида `&x` (где x - цифра в шестнадцатеричной системе счисления).

cr_newline: true - символы LF (\n) заменяются на CR (\r);
crlf_newline: true - символы LF (\n) заменяются на CRLF (\r\n);
universal_newline: true - символы CR (\r) и CRLF (\r\n) заменяются на LF (\n).

Приложение F

Упаковка данных

Произвольные данные могут быть представлены в виде двоичного текста. Для этого элементы массива (данные) описывают с помощью форматных строк (набора спецсимволов).

Синтаксис форматных строк:

- Пробелы в теле форматной строки игнорируются;
- Цифра после спецсимвола соответствует количеству элементов, на которые распространяется его действие;
- Символ звездочки (*) после спецсимвола распространяет его на все оставшиеся элементы;
- Спецсимволы sSiIlL могут начинаться с знака подчеркивания или восклицательного знака, означающих что размер типа данных зависит от операционной системы;
- Добавление символов > или < позволяет использовать старший или младший порядки байтов соответственно (l_> или L!<).

Целые числа:

C - 8-битное целое число без знака (unsigned integer или unsigned char);
c - 8-битное целое число со знаком (signed integer или signed char);
S - 16-битное целое число без знака (uint_16t);
s - 16-битное целое число со знаком (int_16t);
L - 32-битное целое число без знака (uint_32t);
l - 32-битное целое число со знаком (int_32t);
Q - 64-битное целое число без знака (uint_64t);
S_ (S!) - целое число без знака минимально возможного размера (unsigned short);
s_ (s!) - целое число со знаком минимально возможного размера (signed short);
I (I_ или I!) - целое число без знака (unsigned integer);
i (i_ или i!) - целое число со знаком (signed integer);
L_ (L!) - целое число без знака максимально возможного размера (unsigned long);
l_ (l!) - целое число со знаком максимально возможного размера (signed long);
N - 32-битное целое число без знака со старшим порядком байтов (для сетей);

n - 16-битное целое число без знака со старшим порядком байтов (для сетей);
V - 32-битное целое число без знака с младшим порядком байтов (VAX);
v - 16-битное целое число без знака с младшим порядком байтов (VAX);
U - кодовая позиция UTF-8 символа;
w - BER-кодированное целое число.

Десятичные дроби:

D, d - число с плавающей точкой двойной точности;
F, f - число с плавающей точкой;
E - число с плавающей точкой двойной точности, с младшим порядком байтов;
e - число с плавающей точкой с младшим порядком байтов;
G - число с плавающей точкой двойной точности, со старшим порядком байтов;
g - число с плавающей точкой со старшим порядком байтов.

Текст:

A - произвольный двоичный текст с удаленными конечными нулями и ASCII пробелами;
a - произвольный двоичный текст;
Z - произвольный двоичный текст заканчивающийся нулем;
B - произвольный двоичный текст (MSB первый);
b - произвольный двоичный текст (LSB первый);
H - шестнадцатеричный текст (hight nibble первый);
h - шестнадцатеричный текст (low nibble первый);
u - UU-кодированный тексту;
M - MIME-кодированный тексту (RFC2045);
m - base64-кодированный текст (RFC2045, если заканчивается 0, то RFC4648);
P - указатель на контейнер (текст фиксированной длины);
p - указатель на текст, заканчивающийся нулем.

Остальное:

@ - интерпретатор пропускает указанное целым числом количество байтов;
X - интерпретатор продвигается вперед на один байт;
x - интерпретатор возвращается назад на один байт.

Приложение G

Форматирование времени

Форматная строка состоит из групп символов вида:
`%[модификатор][размер][спецсимвол]`.

Любой текст, не относящийся к спецсимволам или модификаторам переносится в результат без изменений.

Размер влияет на количество символов в результате. Если размер меньше, чем необходимое количество символов, то он игнорируется интерпретатором.

Модификаторы:

- `-` - ограничение размера игнорируется;
- `_` - для выделения используются пробелы;
- `0` - для выделения используются нули (по умолчанию);
- `^` - результат в верхнем регистре;
- `#` - изменяется регистр символов.

Год:

`%Y` - номер года с веком;

```
Time.local( 1990, 3, 31 ).strftime "Год: %Y" # -> "Год: 1990"
Time.local( 1990, 3, 31 ).strftime "Год: %7Y" # -> "Год: 0001990"
Time.local( 1990, 3, 31 ).strftime "Год: %-7Y" # -> "Год: 1990"
Time.local( 1990, 3, 31 ).strftime "Год: %_7Y" # -> "Год:    1990"
Time.local( 1990, 3, 31 ).strftime "Год: %07Y" # -> "Год: 0001990"
```

`%G` - номер года с веком, считая от первого понедельника;

```
Time.local( 1990, 3, 31 ).strftime "Год: %G" # -> "Год: 1990"
```

`%y` - остаток от деления номера года на 100 (от 00 до 99);

```
Time.local( 1990, 3, 31 ).strftime "Год: %y" # -> "Год: 90"
```

`%g` - остаток от деления номера года на 100 (от 00 до 99), считая от первого понедельника;

```
Time.local( 1990, 3, 31 ).strftime "Год: %g" # -> "Год: 90"
```

`%C` - номер года, разделенный на 100 (20 в 2011);

```
Time.local( 1990, 3, 31 ).strftime "Век: %C" # -> "Век: 19"
```

Месяц:

%b (%h) - аббревиатура названия месяца (три первые английские буквы);

```
Time.local( 1990, 3, 31 ).strftime "Месяц: %b" -> "Месяц: Mar"
```

%B - полное названию месяца;

```
Time.local(1990, 3, 31).strftime "Месяц: %B" # -> "Месяц: March"
Time.local(1990, 3, 31).strftime "Месяц: %^B" # -> "Месяц: MARCH"
Time.local(1990, 3, 31).strftime "Месяц: %#B" # -> "Месяц: MARCH"
```

%m - номер месяца (от 01 до 12);

```
Time.local( 1990, 3, 31 ).strftime "Месяц: %m" -> "Месяц: 03"
```

Неделя:

%U - номер недели (от 00 до 53), считая от первого воскресенья в году;

```
Time.local( 1990, 3, 31 ).strftime "Неделя: %U" -> "Неделя: 12"
```

%W - номер недели (от 00 до 53), считая от первого понедельника в году;

```
Time.local( 1990, 3, 31 ).strftime "Неделя: %W" -> "Неделя: 13"
```

%V - номер недели (номер недели в формате ISO 8601 - от 01 до 53);

```
Time.local( 1990, 3, 31 ).strftime "Неделя: %V" -> "Неделя: 13"
```

День:

%j - номер дня в году (от 001 до 366);

```
Time.local( 1990, 3, 31 ).strftime "День: %j" # -> "День: 090"
```

%d - номер дня в месяце;

```
Time.local( 1990, 3, 3 ).strftime "День: %d" # -> "День: 03"
```

%e - номер дня в месяце;

```
Time.local( 1990, 3, 3 ).strftime "День: %e" # -> "День:  3"
```

%a - аббревиатура дня недели (три первые английские буквы);

```
Time.local( 1990, 3, 31 ).strftime "День: %a" # -> "День: Sat"
```

%A - полное название дня недели;

```
Time.local( 1990, 3, 31 ).strftime "День: %A" # -> "День: Saturday"
```

%u - номер дня недели (от 1 до 7, понедельник - 1);

```
Time.local( 1990, 3, 31 ).strftime "День: %u" # -> "День: 6"
```

%w - номер дня недели (от 0 до 6, воскресенье - 0);

```
Time.local( 1990, 3, 31 ).strftime "День: %w" # -> "День: 6"
```

Час:

%H - час дня в 24 часовом формате (от 00 до 23);

```
Time.local( 1990, 3, 31 ).strftime "Час: %H" # -> "Час: 00"
```

%k - час дня в 24 часовом формате (от 0 до 23);

```
Time.local( 1990, 3, 31 ).strftime "Час: %k" # -> "Час:  0"
```

%I - час дня в 12 часовом формате (от 01 до 12);

```
Time.local( 1990, 3, 31 ).strftime "Час: %I" # -> "Час: 12"
```

%l - час дня в 12 часовом формате, с приставкой-пробелом (от 0 до 12);

```
Time.local( 1990, 3, 31 ).strftime "Час: %l" # -> "Час: 12"
```

%p - индикатор меридиана (”AM” или ”PM”);

```
Time.local( 1990, 3, 31 ).strftime "%I %p" # -> "12 AM"
```

%P - индикатор меридиана (”am” или ”pm”);

```
Time.local( 1990, 3, 31 ).strftime "%I %P" # -> "12 am"
```

Минуты и секунды:

%M - количество минут (от 00 до 59);

```
Time.local( 1990, 3, 31 ).strftime "мин: %M" # -> "мин: 00"
```

%S - количество секунд (от 00 до 60);

```
Time.local( 1990, 3, 31 ).strftime "сек: %S" # -> "сек: 00"
```

%N - дробная часть секунд.

- *%3N* - миллисекунды;
  %6N - микросекунды;
  %9N - наносекунды (по умолчанию);
  %12N - пикосекунды.

%L - количество миллисекунд (от 000 до 999);

```
Time.local( 1990, 3, 31 ).strftime "мс: %L" # -> "мс: 000"
```

%s - количество секунд, прошедших начиная с 1970-01-01 00:00:00 UTC;

```
Time.local( 1990, 3, 31 ).strftime "%s" # -> "638827200"
```

Форматы:

%D - форматная строка `"%m/%d/%y"`;

```
Time.local( 1990, 3, 31 ).strftime "Дата: %D"
# -> "Дата: 03/31/90"
```

%F - форматная строка `"%Y-%m-%d"` (время в формате ISO 8601);

```
Time.local( 1990, 3, 31 ).strftime "Дата: %F"
# -> "Дата: 1990-03-31"
```

%v - форматная строка `"%e-%b-%Y"` (время в формате VMS);

```
Time.local( 1990, 3, 31 ).strftime "Дата: %v"
# -> "Дата: 31-MAR-1990"
```

%c - формат системы;

```
Time.local( 1990, 3, 31 ).strftime "Система: %c"
# -> "Система: Sat Mar 31 00:00:00 1990"
```

%r - форматная строка `"%I:%M:%S %p"`;

```
Time.local( 1990, 3, 31 ).strftime "%r" # -> "12:00:00 AM"
```

%R - форматная строка `"%H:%M"`;

```
Time.local( 1990, 3, 31 ).strftime "%R" # -> "00:00"
```

%T - форматная строка `"%H:%M:%S"`;

```
Time.local( 1990, 3, 31 ).strftime "%T" # -> "00:00:00"
```

Форматирование:

%n - перевод строки;

```
Time.local( 1990, 3, 31 ).strftime "%D %n %F %n %c"
# -> "03/31/90 \n 1990-03-31 \n Sat Mar 31 00:00:00 1990"
```

%t - отступ;

```
Time.local( 1990, 3, 31 ).strftime "%D %t %F %t %c"
# -> "03/31/90 \t 1990-03-31 \t Sat Mar 31 00:00:00 1990"
```

Остальное:

%x - только дата;

```
Time.local( 1990, 3, 31 ).strftime "%x" # -> "03/31/90"
```

%X - только время;

```
Time.local( 1990, 3, 31 ).strftime "%X" # -> "00:00:00"
```

%z - смещение часового пояса относительно UTC;

```
Time.local( 1990, 3, 31 ).strftime "%z" # -> "+0400"
```

- *%:z* - часы и минуты разделяются с помощью двоеточия;

  ```
  Time.local( 1990, 3, 31 ).strftime "%:z" # -> "+04:00"
  ```

%::z - часы, минуты и секунды разделяются с помощью двоеточия;

```
Time.local( 1990, 3, 31 ).strftime "%::z" # -> "+04:00:00"
```

%Z - название временной зоны;

```
Time.local( 1990, 3, 31 ).strftime "%Z" # -> "MSD"
```

%% - знак процента.

Приложение H

Создание потоков

Вид потока (mode) устанавливается с помощью группы модификаторов.

Модификаторы:

r - только для чтения;
r+ - как для чтения, так и для записи (новые данные вместо старых);
w - только для записи (новые данные вместо старых). При необходимости создается новый файл;
w+ - как для чтения, так и для записи (новые данные вместо старых). При необходимости создается новый файл;
a - только для записи (новые данные добавляются к старым). При необходимости создается новый файл;
a+ - как для чтения, так и для записи (новые данные добавляются к старым). При необходимости создается новый файл;
b - двоичный режим (может использоваться с другими модификаторами). Чтение из файла выполняется в ASCII кодировке. Используется только для чтения двоичных файлов в Windows;
t - текстовый режим (может использоваться с другими модификаторами).

При чтении или записи данных используются две кодировки: внутренняя и внешняя.

- Внешняя кодировка - это кодировка текста внутри потока. По умолчанию она совпадает с кодировкой ОС.
- Внутренняя кодировка - это кодировка для работы с полученными данными внутри программы. По умолчанию она совпадает с внешней кодировкой.

Если внутренняя и внешняя кодировка отличаются, то при чтении данных выполняется автоматическое преобразование из внешней кодировки во внутреннюю, а при записи данных - из внутренней во внешнюю.

После модификатора могут быть указаны внешняя и внутренняя кодировки, разделенные двоеточием: `"w+:ascii:utf-8"`. Если внутренняя кодировка не указана, то по умолчанию используется внешняя кодировка.

Вид потока также может быть изменен с помощью дополнительного аргумента - массива опций или набора констант из модуля File::Constants.

Опции:

mode: вид создаваемого потока;
textmode: true, текстовый режим;
binmode: true, двоичный режим;
autoclose: true, закрытие файла, после закрытия потока;
external_encoding: внешняя кодировка;
internal_encoding: внутренняя кодировка. Если внутренняя кодировка не указана, то по умолчанию используется внешняя кодировка;
encoding: внешняя и внутренняя кодировки в формате `external:internal`.

Также принимаются элементы, влияющие на преобразование кодировок.

Приложение I

Файловая система Linux

Файл - это фундаментальная абстракция в Linux (практически любая сущность считается файлом). Операции с файлами осуществляются с помощью файловых дескрипторов (цифровых идентификаторов).

I.1. Типы файлов

- *Обычный файл* - файл, позволяющий вводить или выводить данные, а также перемещаться по ним с помощью буферизации (сохранения данных в буфер);
- *Жесткая ссылка* - файл, ссылающийся на другой файл.

  Жесткие ссылки необходимы, чтобы использовать нескольких имен для одного файла. При этом все жесткие ссылки равноправны - файл будет удален только в том случае, если удалены все жесткие ссылки на него.

  Изменение любой жесткой ссылки повлияет на связанный с ней файл.

  Создание жестких ссылок возможно только на одном физическом носителе информации (жестком диске, карте памяти и т.д.);
- *Символьная ссылка (ярлык)* - файл, ссылающийся на другой файл.

  Ярлыки необходимы, чтобы использовать нескольких имен для одного файла. При этом существует одно основное имя - файл будет удален только в том случае, если удален основной файл.

  1. Изменение любой символьной ссылки повлияет только на саму ссылку.
  2. Изменение основного файла повлияет на все связанные с ним ссылки;

- *Блочное устройство* - файл, обеспечивающий интерфейс доступа к какому-либо устройству.

  Ввод и вывод данных в блочные устройства выполняется в виде блоков, размер которых устанавливается блочным устройством. При этом существует возможность перемещаться по данным в пределах блочного устройства. Жесткий диск - это один из примеров блочных устройств;
- *Символьное устройство* - файл, обеспечивающий интерфейс доступа к какому-либо устройству.

  Ввод и вывод данных в символьное устройство выполняется в виде отдельных байтов. Обычно ввод и вывод данных не буферизуется и не существует

возможности перемещаться по данным в пределах символьного устройства. Терминалы или модемы - это примеры символьных устройств.

- *Сокет* - файл, обеспечивающий коммуникацию между различными процессами, которые могут выполняться на разных компьютерах.

I.2. Поиск файлов

Поиск файлов выполняется с помощью путей.

- Путь в Ruby - это текст или любой другой объект, отвечающий на вызов метода `object.to_path`.
- Путь в файловой системе - это полный список имен каталогов, (заканчивающийся именем самого файла), по которому может быть найден файл.

Виды путей:

- *Абсолютный путь* - путь к файлу, начинающийся от корневого каталога или буквы диска;
  Относительный путь - путь к файлу, относительно текущего каталога.

Спецсимволы:

- Для перемещения на один каталог вверх, используется . .;
- Для обозначения текущего каталога используется .;
- Для обозначения домашнего каталога используется ~;
- Для разделения каталогов в Windows используется символ обратной косой черты (\), а в Linux - символ косой черты (/);
- Расширение файла отделяется точкой от его имени.

* - соответствует любому файлу или любой группе символов в имени файла;
** - соответствует любому каталогу в имени файла (включая символ разделителя);
? - соответствует любому одиночному символу в имени файла;
[...] - соответствует любому одному символу в имени файла из указанных в квадратных скобках;
[^...] - соответствует любому одному символу в имени файла, кроме указанных в квадратных скобках;
{ } - логическое или между символами, разделенными запятыми (Ruby 2.0).

Также существует набор констант, влияющих на поиск.

Константы:

File::FNM_SYSCASE - чувствительность к регистру зависит от ОС (действует по умолчанию);
File::FNM_CASEFOLD - игнорирование регистра;
File::FNM_PATHNAME - спецсимвол ?, не будет соответствовать косой черте (символу разделителя);
File::FNM_NOESCAPE - обратная косая черта будет соответствовать самой себе, а не экранировать следующий символ;
File::FNM_DOTMATCH - знак точки в имени файла будет считаться частью имени, а не разделителем для расширения (используется для поиска скрытых файлов в Linux);
File::FNM_EXTGLOB - фигурные скобки в имени файла будут содержать выражение логического ИЛИ (Ruby 2.0).

Использовать несколько констант можно с помощью выражения побитового ИЛИ. Тем, кто хочет разобраться почему это работает, следует изучить двоичную арифметику.

I.3. Доступ к файлам

В некоторых файловых системах предусмотрена возможность ограничения доступа пользователей к содержимому файла. При этом обычно выделяют три типа прав: право на чтение, право на запись и право на выполнение.

Права доступа устанавливаются с помощью четырех целых чисел. Каждая цифра соответствует двоичному биту, добавляемому к файлу. Права доступа устанавливаются для определенного пользователя, определенной группы и для всех пользователей (идентификаторы пользователя и группы устанавливаются отдельно).

- Первая цифра считается дополнительной и определяет либо различные способы запуска файла, либо дополнительное условие для каталогов;
- Оставшиеся три цифры объявляют права доступа для пользователя, для группы и всех остальных пользователей соответственно.

Данные цифровые коды могут быть применены и в Windows. При этом допускается ограничивать доступ только для чтения и записи информации.

Основные права (permission):

700 - пользователь имеет право на чтение, запись и выполнение;
400 - пользователь имеет право на чтение;
200 - пользователь имеет право на запись;
100 - пользователь имеет право на выполнение (или право на просмотр каталога);
70 - группа имеет право на чтение, запись и выполнение;
40 - группа имеет право на чтение;
20 - группа имеет право на запись;
10 - группа имеет право на выполнение (или право на просмотр каталога);
7 - все остальные пользователи имеют право на чтение, запись и выполнение;
4 - все остальные пользователи имеют право на чтение;
2 - все остальные пользователи имеют право на запись;
1 - все остальные пользователи имеют право на выполнение файла (или право на просмотр каталога).

По умолчанию файл выполняется от имени того пользователя, которым файл был запущен.

Дополнительные права:

4000 - файл запускается от имени владельца;
2000 - файл запускается от имени установленной группы пользователей;
1000 - из каталога можно удалить только те файлы, владельцем которых является пользователь.

Права доступа определяются после сложения чисел, объявляющих доступ для отдельных категорий пользователей.

0444 - любой пользователь имеет право только на чтение информации из файла.

Привилегии пользователей определяются с помощью цифровых идентификаторов. Каждый пользователь имеет четыре вида идентификаторов;

- *UID* - реальный идентификатор пользователя, запустившего файл;
  EUID - действующий идентификатор пользователя, с которым файл выполняется;
  GUID - реальный идентификатор группы, к которой относится пользователь, запустивший файл;
  EGUID - действующий идентификатор группы пользователей, с которым файл выполняется;

Для проверки привилегий пользователя обычно используется действующий идентификатор.

Доступ к файлу также может быть ограничен с помощью набора констант, передаваемых при открытии файла.

I.4. Открытие файла

Константы:

File::RDONLY - файл открывается только для чтения;
File::WRONLY - файл открывается только для записи;
File::RDWR - файл открывается как для чтения, так и для записи;
File::APPEND - запись данных в конец файла (разрешение на запись необходимо устанавливать отдельно);
File::CREAT - создание нового файла. Определение владельца выполняется по действующему идентификатору. Определение группы выполняется по идентификатору группы для программы или для базового каталога;
File::DIRECT - ограничение кэширования содержимого файла;
File::EXCL - ограничение возможности создания ярлыков;
File::NONBLOCK - если система не готова выполнить запрос к файлу немедленно, то процесс выполнения не блокируется, а получает сигнал от системы;
File::TRUNC - новые данные сохраняются вместо существующих (разрешение на запись необходимо устанавливать отдельно);
File::NOCTTY - терминал открывается, но управление программой не передается;
File::BINARY - двоичный режим;
File::SYNC, File::DSYNC, File::RSYNC - синхронное открытие файла. При записи в поток, он будет блокировать процесс выполнения до тех пор, пока информация не будет реально записана на устройство;
File::NOFOLLOW - открываются сами ярлыки, а не связанные с ними файлы;
File::NOATIME - время последнего доступа не обновляется.

Заключение

Лучший способ разобраться в чем-то до конца - попробовать научить этому компьютер.

Дональд Кнут

Вот и всё, что я смог найти. Не думайте, что прочитав эту книгу вы сразу станете писать высоко-нагруженные приложения. Максимум чему вы научились - это программирование небольших скриптов, способных немного облегчить вашу повседневную работу. Еще множество необходимых знаний о стиле кода, тестировании и отладке, архитектуре и оптимизации (и т.д.) отделяет вас от гордого звания программиста. Могу лишь надеяться, что удовольствия от работы с Ruby поможет преодолеть все эти препятствия и сообщество получит еще одного единомышленника.

Как вы могли заметить в этой книге приведено очень мало примеров. И это не случайно. Изучение исходного кода уже работающих и востребованных приложений поможет вам быстрее понять все особенности и возможности Ruby. Сделав свой вклад в развитие какой-либо существующей программы вы не только улучшите свои навыки по написанию кода, но также поможете развитию языка. Любые же примеры, которые могут быть приведены в книге изначально очень сильно ограничены.

Дополнительную информацию о Ruby, можно найти:

http://www.google.com - здесь есть все что вам необходимо;

http://www.rubygems.org - хранилище пакетов (использующих Ruby и менеджер пакетов RubyGems);

http://www.ruby-toolbox.com - удобный каталог, классифицирующий существующие gem-ы (пакеты);

http://github.com - сайт для публикации исходного кода. В нем есть раздел и для Ruby-программистов.